SymPLe 1131: A Novel Architectural Solution for the Realization of Verifiable Digital I&C Systems and Embedded Digital Devices

Carl Elks, Matt Gibson, Tim Bakker, Richard Hite, Smitha Gautham, Vidya Venkatesh, Jason Moore

Abstract

One of the main reasons for the increasing popularity of FPGAs in embedded systems and digital I&C systems is that they provide many of the advantages from both hardware and software components. FPGAs provide configurability approaching that of SW, while at the same time delivering deterministic behavior of pure hardware solutions. However, with regard to Software Common Cause Failures (CCF), the use of FPGAs in digital I&C platforms should not be seen as plug-in remedy for reduction or elimination of CCF.

To address CCF, we argue that it is not enough to manage the complexity of an I&C device or system, but we must also have the ability to constrain complexity (as needed) and rigorously reason about execution and device functionality. This paper presents the work done regarding an FPGA overlay architecture called SymPLe. SymPLe is designed to reduce the gap between software and hardware approaches for digital I&C systems and increases deterministic behavior and verifiability of the application, architecture and safety-critical system while reasoning about CCF.

1 Introduction

Most nuclear power plants built in the 1970s and 80s have analog Instrumentation and Control (I&C) systems from that era. Although that technology is robust, mature and reliable it lacks some of the advantages of modern digital systems. Furthermore, the technology has exceeded its life expectancy and is becoming obsolete. Digital I&C systems are more precise, easier to replace and have additional functionalities and diagnostic capabilities that its analog counterparts cannot provide.

Although the integration of digital technology in systems with stringent safety-critical requirements has been slow it has not been without success. The aviation industry, with the FAA being a key player, has seen successful in integrating digital technology throughout critical and non-critical systems with publishing several guidelines for electrical (DO-254) and software (DO-178 B/C) airborne systems. Although other industries have been early adopters, the nuclear power industry has been somewhat conservative in its exploration of digital systems, mainly due to concerns regarding Common Cause Failures (CCF) and in particular software CCF (SCCF).

The conventional design strategy against CCF involves diversity and defense in depth creating multiple layers and boundaries for an event to become a safety-critical failure. Although this design approach is fundamental to current safety-critical systems designs it has significant disadvantages. Implementation of diversity can be applied at all levels and different stages. NUREG/CR-6303 lists six different attributes: Design, Equipment, Functional, Human, Signal, and Software [1]. Large scale diversity and defense in depth strategies are costly, complex, require increased plant integration and infer high qualification and validation costs. The burden and cost associated with verification and qualification of digital I&C needs to be well addressed and investigated in order to meet regulatory requirements. The research and results presented in this paper propose a novel architectural approach, called SymPLe, addressing architectural decisions and principles reducing the verification and validation burden.

1.1 Fundamentals of SymPLe

The current digital I&C market is mostly based on general purpose processors and micro-controllers not suitable per definition for safety-critical applications. We propose an alternative that allows end-users to: (1) support verification and validation (V&V) from inception and (2) constrain complexity to the bare necessities for the application.

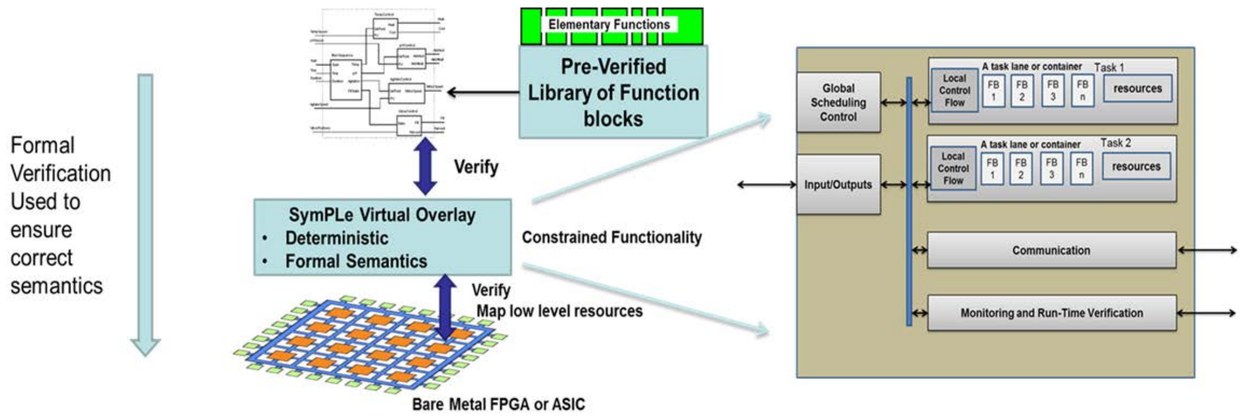


Figure 1: High level perspective of the SymPLe Architecture Concept
.

Currently, a big shift can noticed towards employing FPGA's as an alternative for the general processor based embedded system. Although this shift considerably reduces complexity and eliminates unneeded functionality it will not address CCF or design flaws adequately.

The SymPLe architecture provides the capability to reason about functionality and complexity, and to create a custom execution platform to support safety-critical applications. The end-user is not only guided by a design process but is also provided an execution architecture that has been formally verified from execution semantics, low-level FPGA or ASIC representation to hardware in the loop validation. SymPLe is specifically aimed for the nuclear industry with its stringent safety requirements but fairly modest functionality and logic. The principle motivation for the SymPLe architecture is to provide a V&V-aware execution architecture or device to host critical plant functions while reducing the cost of qualifying these devices for use in Nuclear Power Plants. The fundamentals for SymPLe are:

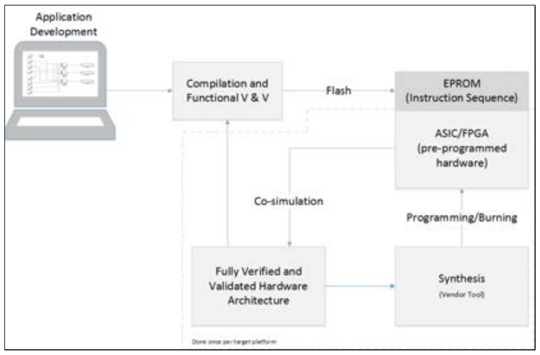
- **Complexity-Aware:** minimize complexity to reduce the validation and qualification cost
- **Model based design:** use model-based engineering to apply formal verification and testing techniques
- **Transparent Verification:** a transparent architecture to ease the burden of testing and verification

1.2 Overlay Architecture

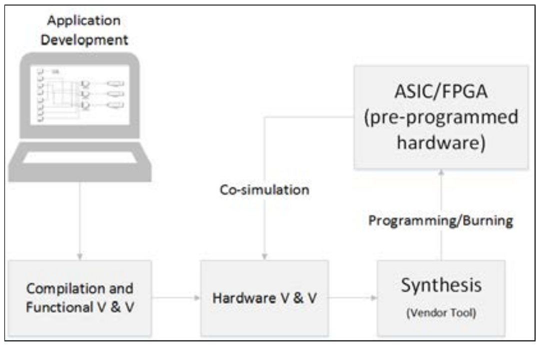
The concepts applied within the SymPLe architecture are strongly based on the IEC-61131 standard for programming Programmable Logic Controllers and in particular the use of Function Blocks as outlined IEC-61499. The nuclear industry has a strong history with using PLCs and it is our belief that the SymPLe architecture can apply this strong relationship to ease the transition and introduction of SymPLe based digital I&C devices. The IEC-61131 standard for programming PLCs does not only influence the basic architecture of the SymPLe architecture and its operational semantics but also the nature of defining and creating the application functionality. The SymPLe architecture and its final implementation on the device (e.g. FPGA or ASIC device) is represented as an overlay architecture on top of the device intrinsic architecture, creating an abstracted, controlled and contained execution environment. Many developed overlay architectures [2] have been for the purpose of portability whereas the SymPLe architecture is developed to support a constrained execution environment for IEC-61131 inspired function blocks. SymPLe is designed to enforce deterministic and predictable behavior where the operational semantics are being formally verified to ensure high trust in functional correctness. Figure 1 is an overview of how SymPLe is positioned on top of the bare metal of the FPGA and acts as an abstracted execution environment for the higher layer function block application. By applying formal verification methods at the model-based and hardware description language (HDL) of the device, to applying conventional testing and co-simulation methods, SymPLe builds up strong evidence for the safety-assurance case and prevention of CCF.

1.3 Implementation Workflow

The SymPLe architecture lends itself to be used within several different workflows. As envisioned, an application engineer within a development environment would have a library of supported function blocks and add-on or macro function blocks to create desired applications. Once the application has been completed a custom compiler and verification tool would create a sequence of SymPLe supported generic function blocks. From here two independent tracks could be observed. On one track the sequence will generate a custom SymPLe architecture limited to only the required application functionality and a subset of the SymPLe architecture function blocks. In the second track the sequence is fed to a device already pre-programmed with a full SymPLe architecture where the sequence will invoke the pre-existing function blocks in the correct order. Figure 2 depicts these two workflows.


Application Development
Compilation and Functional V & V
Flash
EPROM (Instruction Sequence)
ASIC/FPGA (pre-programmed hardware)
Co-simulation
Programming/Burning
Fully Verified and Validated Hardware Architecture
Synthesis (Vendor Tool)
Done once per target platform

(a) Pre-progammed SymPLe device



(b) Application dependent hardware

Figure 2: Workflow tracks

2 Related Work

Virtualizing hardware is a common method for the information technology sector and is widely applied on workstations, servers, and high-performance supercomputers. Recently, virtualization has moved from being used only on micro-processor systems to virtualizing FPGA architectures, also denoted as FPGA overlays [2]. Motivations for creating an FPGA overlay architecture are: overcoming limited hardware resources, enriching the capabilities of the existing FPGA, and providing a consistent uniform execution platform across different families and vendors of FPGAs [3].

A virtual FPGA platform is explored in [4] where control system software for PLCs is converted to an intermediate code called Virtual Machine Assembler (VMASM). Two virtual FPGA execution machines (basic and enhanced) have been designed with the ability to execute a wide range of function blocks on IEC-61131 defined data types. The basic machine used 8-bit wide data buses while the enhanced version provides 32-bit wide data buses. In contrast to the SymPLe architecture that is not optimized for speed but is tailored towards safety critical systems, the presented virtual machines were benchmarked against a general purpose micro-controller and achieved an execution speed-up of 80 to 180.

Chmiel et al. in [5] present a virtualized PLC architecture that implements a subset of the function blocks outlined in IEC-61131-3. The architecture follows a conventional computer system with a central processing unit, memory for holding data, and I/O controllers. The architecture supports integer and real number types and includes specialized DSP functional blocks for optimized and complex arithmetic operations. An Application Specific Instruction Set processor was presented in [6]. The processor is in compliance with the IEC-61131-3 instruction list and implements many of the functional units generally found in a computing system.

Other affiliated research can be found in translating PLC based control programs to synthesizable Hardware Description Language representation. A systematic approach for converting PLC based control programs is shown in [7]. As an intermediate step the research applies Enhanced Data Flow Graphs to capture the essence of the control program and applies optimization and simplification techniques to reduce the number of nodes and edges in the graph. Economakos et al. have built a tool set for converting PLC programs into C code, and subsequently translating this into VHDL from where the vendor synthesis tools can create FGPA netlists [8]. The methodology has been applied to three typical control algorithms: a PID, a fuzzy controller, and an adaptive fuzzy controller.

Shukla et al. in [9–12] have proposed QUKU, a course grained reconfigurable architecture to bridge the gap between general purpose microprocessors and single-purpose ASICS. The architecture enables the possibility for easy reconfiguration based on the intended application, through the use of a reconfigurable array of Processing Elements (PE).

A micro-kernel hypervisor was introduced in [13] for a hybrid FPGA architecture with an ARM Dual Core processor tightly coupled to a FPGA fabric. The authors introduce an intermediate fabric which consist of a set of processing elements interconnected by crossbar switches where the hypervisor controls communication and functionality of the reconfigurable FPGA regions.

In contrast to the above presented research, SymPLe is defined by constrained architectural complexity, unique use-case formulation, and rigorous formal and runtime verification methods applied. From formally verifying and validating the application correctness, the architectural model, and the HDL translation, to extensive co-simulation on the target device, SymPLe achieves strong arguments for safety assurance and elimination of Software CCF.

3 SymPLe Architecture

The SymPLe architecture and its design decisions are based upon three principles: simplicity, verifiability, and determinism. These principles have guided the architecture to be consistent across all layers of the design providing a clean and light architecture. SymPLe possesses the strong attributes of formal operational semantics, transparency, and clear separation between control and data flow throughout the architectural layers. A paramount requirement for the safety-critical systems targeted for SymPLe applications is the assurance of completely deterministic behavior, and SymPLe provides such a guarantee. Lastly, for ease of verification complexity and functionality is evenly distributed throughout the layers of the SymPLe architecture.

3.1 Architectural Organization

SymPLe is a unique architecture hierarchical in nature where complexity is evenly introduced at all levels and subcomponents. This even spreading of complexity increases modularity and increases the ease of reasoning in verification and validation efforts at all stages and components in the system. As can be seen in Figure 1, the architectural model is distinct and does not resemble anything close to the Von Neumann or Harvard architecture of a typical computer system. At the application level, SymPLe provides the ability for true parallelism of task execution due to multiple localized containers or task lanes as realized in either a FPGA or an ASIC.

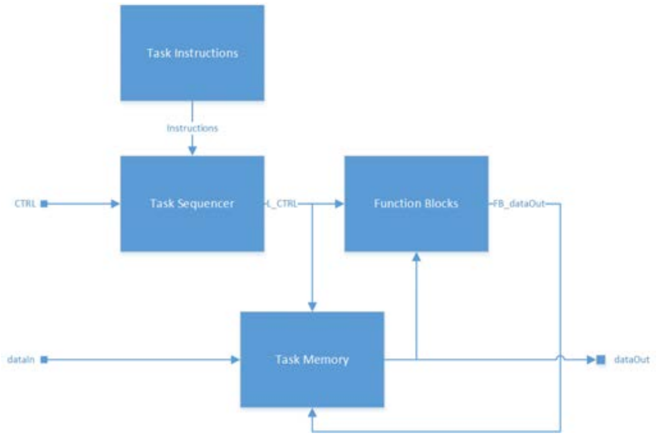


Figure 3: Task-level Design

.

Each task container embeds a full or selectively chosen set of function blocks as required by the specific task or application, a task sequencer, and a task memory, shown in Figure 3. Task priorities and frequencies are controlled by a global scheduler or sequencer, synchronizing task execution and moving data between tasks, I/O, and communication

Table 1: SymPLe supported Function Blocks

Instruction	Description
AND, OR, NOT, XOR, NAND, NOR	Logical operators
AND, OR, NOT, XOR, NAND, NOR	Bit-wise operators
MAX, MIN, MUX	Selection operators
GT, GE, EQ, LT, LE, NE	Comparison operators
ADD, SUB, MUL, DIV	Arithmetic operators
SL_BIT, SR_BIT	Bit-shift operators
MOVE	System operator

channels. The task sequencer orchestrates execution of the function blocks in its lane in sequential order thereby resolving the management problem of data from and to the function blocks. Unique specialized tasks can be added to the SymPLe architecture for handling specific problems. One example of a specialized task would be a high-speed data communication block typically not suitable for implementation in one of the function block task lanes. Another type of task lane involves verifying and checking stringent run-time properties at all levels within the architecture such as divide by zero detection in functions blocks and insurance of data integrity at the task level.

Two design variants of SymPLe were initially explored, where the major difference between the two was the complexity within individual function blocks. In one variant of SymPLe, the autonomous function block, the function block was self-sufficient in that it knew when to perform an operation and where to fetch and store its data, all on its own accord. In contrast, the lite function block variant employed function blocks stripped down to its essentials and required regulation by a task sequencer. Due to increased complexity with little justification, development of the autonomous function block variant was ceased and the lite variant was chosen for future efforts. Table 1 lists the hardware supported function blocks SymPLe; which is a "functionally complete" set of operators. Figure 4 depicts the method in which multiple function blocks are assembled into a single construction.

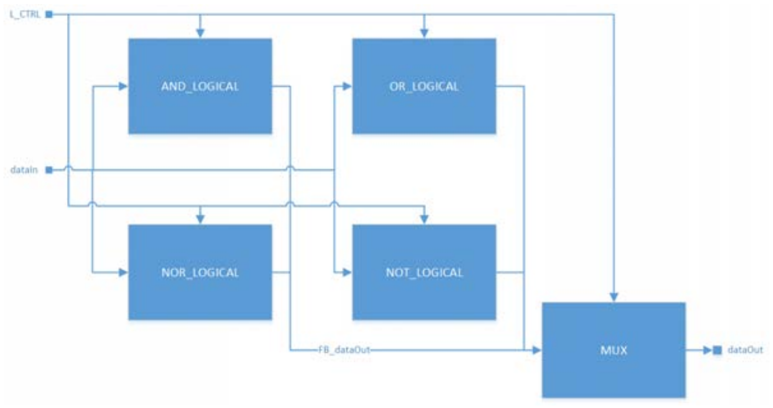


Figure 4: Function Block Level Design

.

3.2 Data Representation and Arithmetic Computation

The typical I&C application, within the existing group of operational Light Water Reactors, have little dependency on true floating point operation as defined by IEEE 754-2008. SymPLe, within its current design cycle does not incorporate floating point numbers but is solely dependent on the well-mannered and easy characteristics of 32-bit fixed-point signed numbers. A $Q_{m.n}$ format is used to describe the integer and fractional portions of the 32-bit data within a container, where the number of fractional bits n are determined based on the functionality and data operations within the task.

3.3 Operational Semantics

A fundamental decision in developing formal semantics for function blocks is to determine what model of computation or language is best suited to guarantee deterministic and predictable behavior. At present, we are focusing on models from reactive system theory [14]. Reactive systems are based on synchronous behaviors paradigm. A reactive system is characterized by its ongoing interaction with its environment, continuously accepting requests from the environment and continuously producing results. This type of model of computation is well suited to real time continuous control and monitoring systems which are typical of nuclear control and monitoring functions. 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 synchronous model has been largely adopted by the automation community (e.g. LabVIEW, Mathworks Simulink, and Scade Lustre). The synchronous postulates for the SymPLe function blocks are:

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

The scan-cycle as it is defined for SymPLe is the unified repetition period, in Clock Cycles (CC's), and includes the capturing of inputs, execute all the tasks to completion (dominated by longest task) and update the outputs. The scan-cycles, because of the deterministic behavior of SymPLe, can be calculated using 1.

$$S = (\alpha \cdot max\{s_n\}) + (\beta \cdot d) + C \qquad \forall t_n \in T \tag{1}$$

Where S is the scan cycle in CC and α is the number of CC's required for executing a single function block ($\alpha = 13 \quad CC$). As mentioned in the previous chapter, the task with the most sequence steps s_n out of a set of tasks T, is determining the scan cycle time. β is the number of CC's used for executing a set of operations d which defines capturing inputs, setting outputs and transferring data between tasks. The constant C is additional clock cycles for switching between executing function blocks and performing the capture and output operations, and takes 5 CC's.

3.3.1 Function Blocks

Operational semantics for function blocks are well defined and can be best described as a finite state-machine, waiting in an idle state. A trigger event will start execution of a function block and will start clocking in all the input data into a series of input shift registers. Subsequently, once all data has been clocked in, some run-time properties are verified and ensured to guarantee correct execution behavior. Combinational logic operates on the data in the shift register calculating the result to achieve the desired functionality. The result is then again verified and stored in an output register.

The function blocks execute with neither knowledge of the origin of the data to which it operates on nor of the final destination in memory in which the result of the function will reside in memory. The function blocks receive data at a coordinated time, referenced to the function block trigger, calculates the result, and stores it in its output register(s).

Function block (FB) models are partitioned into four separate sections. The four pieces include Control, Data, Runtime Verification, and Design Verification. FBs follow the model established by standard IEC-61499 where each function block has separate control flow and data flow. The controller portion receives a trigger(s) to execute, determines the state of the function block, enables data registers, and asserts the done signal. All data is routed through the data flow portion of a function block. The data flow portion consists of input registers, combinational logic for functional operations, and output registers. The registers in this partition are controlled by the function block controller.

The runtime verification partition is synthesized into hardware along with the control and data partitions. It performs verification of the operation of the function block during runtime. Examples of runtime verification include type checking and overflow detection. The final partition of a function block is the Design Verification piece. This part encases temporal properties of the function block that are verified at the model level during design. These properties

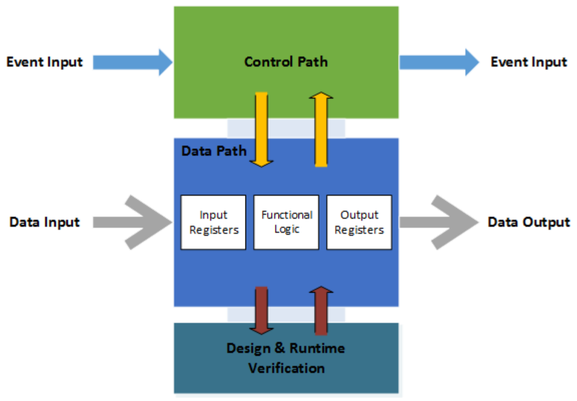


Figure 5: Function Block operational schematic

.

are not synthesized into hardware but are used in verification software that uses formal methods to ensure that no violation of these properties exists mathematically. The properties in the Design Verification section apply to either the controller or the data flow portion of a function block. An example of a property for the control flow of the Lite variant is: "After the trigger goes high, 2 cycles later the shift registers will be clocked."

3.3.2 Local Sequencer

The task sequencer, shown in Figure 3, coordinates the execution of task instructions. Beginning in the initialization stage shown in Figure 6, the task sequencer waits for a trigger from the global sequencer to begin its execution. Once this trigger is received, the task sequencer begins execution of the task instructions by starting the execution of the first function block in the task sequence. The task sequencer signals to the task memory which memory location to fetch and the activated function block reads this data from its data line. The read data state cycles until all data inputs are read and registered inside of the activated function block. Once all data is read, the function block is executed and the result of the function block is written to a task memory location as determined by the task sequencer. The task sequencer continues activating function blocks in this manner until all instructions in the task sequence are exhausted. The global sequencer will then reset the trigger and the task sequencer will re-enter the initialization state in which it will remain until again triggered by the global sequencer.

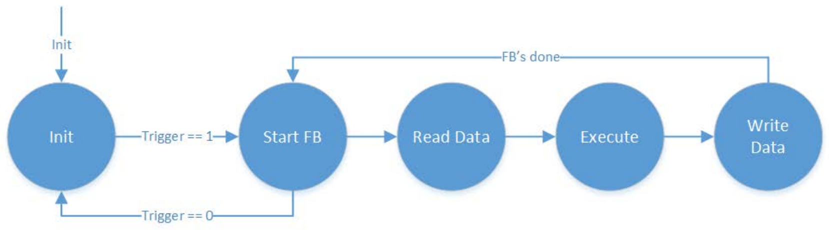


Figure 6: Task Sequencer State Machine

.

3.3.3 Global Sequencer

The global sequencer resides in the top-level of the SymPLe architecture and has two execution stages or states as shown in Figure 7. During the preload stage instructions are executed by the global sequencer that load task memories with data literals. Preload instructions are a 4-tuple consisting of a task number, task memory address, data, and an end of instruction flag. For example, during the preload sequence the constants zero and one are loaded into each task memory for use in logical operations by that task. Another operation occurring during the preload stage is storing the initial $Q_{m.n}$ format in each individual task memory. The global sequencer cycles through all preload instructions in the global sequence until these values are stored in the relevant task memory locations. Once the cycle is complete the global sequencer transitions to the next state, the done sequence.

The next state for the global sequencer is the done sequence, shown in Figure 7. In this state the global sequencer moves data from IO to a task, from one task to another task, or from a task to IO. During this state the global sequencer cycles through all instructions that require global movement of data. These instructions are a 4-tuple consisting of: 1) the task number, or IO, to read from, 2) the address in which the data resides, 3) the task number, or IO, to write the data to, and 4) the final address in which to store the data. Once the done sequence is complete, the global sequencer triggers tasks to execute on the data stored in their task memories. When complete, each task will signal that it is done and the global sequencer will begin again its done sequence by initiating the required global data transfers. The done sequence continues to be executed in this fashion until the architecture is de-energized.

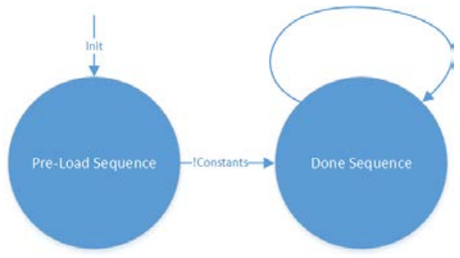


Figure 7: Global Sequencer State Machine

.

4 Verification

SymPLe has been developed around an architecture that is intended to be modular by design. By constructing the architecture in this manner, it opens the potential for validating and verifying portions of the design at every component level and at the system level as well. This transparent effort in verification at all levels is a fundamental strength of SymPLe.

4.1 Model Verification

To validate an architecture like SymPLe, there are a variety of methods to develop a verification strategy. One of the most popular methods is to build up a variety of functional tests based upon project requirements. This relies on the developer or quality engineer to build up a suite of tests that fully exercise an algorithm. A more rigorous verification approach is to use Simulink Design Verifier. Simulink Design Verifier provides a feature known as property proving to formally prove the algorithmic behavior of a model meets specified requirements. Design Verifier allows users to construct proofs of properties based on requirements inside of a model. Based upon these proofs, Simulink Design Verifier will search a constrained space to seek out any violations to the proof. When a violation is detected, Simulink Design Verifier provides a counter example test case where the violation occurred in order to debug the violation. Next, the user will either update the proof or the algorithm model.

Ideally, using Simulink Design Verifier would require few modifications to an initial design. However, in practice there is a set of guidelines that can be followed that improve the speed of analysis and the outcome of the analysis. The following steps are recommended as best practices:

1. Run design error detection in an attempt to find any design errors such as divide by zero
2. Assign port data types
3. Specify port minimum/maximum values
4. Attempt to remove all unnecessary floating point data types
5. Execute property proving in find violation mode

6. Run property proving on the smallest proofs working up to the entire system
7. Switch property proving mode to prove for the entire model

During verification of a model, the possibility exists that there are portions of the model where a particular proof is unsolvable. Typically with this outcome the root cause is the complexity and size of the mathematical space that Simulink Design Verifier must search. If it is not possible to further constrain the search space, the next recommended course of action is to stub these portions of the model out of the analysis. Alternatively, if a section of a model was found to be unsolvable, these sections of the model may be verified using traditional functional test cases. One goal of our verification work flow is to leverage property proving capabilities to minimize the reliance on functional testing.

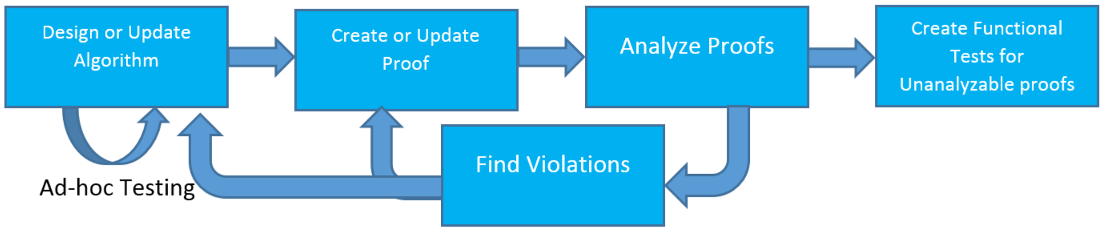

Design or Update Algorithm
Create or Update Proof
Analyze Proofs
Create Functional Tests for Unanalyzable proofs
Find Violations
Ad-hoc Testing

Figure 8: Proposed work flow

.

The block diagram, in figure 8 details the verification work flow discussed. By utilizing this work flow, it is possible to reduce the number of necessary test cases and shift more of the validation efforts to solving proofs. In addition to having full verification efforts on the model-based design of the SymPLe architecture the project also utilizes property proving on the HDL implementation, generated from the model-based SymPLe design.

4.2 HDL Verification

Mentor Graphics Questa model checker was used to provide verification at the lower levels of the design (the VHDL code) - typically synthesized on a FPGA or ASIC. The objective of this effort was to explore if these two formal verification tools can be connected to provide end-to-end coverage. To achieve this, we use a method called Assertion Based Verification (ABV) in which the properties of a system are written in form of assertions (executable statements). Questasim uses Property Specification Language (PSL) or System Verilog Assertion to define a type of formal temporal logic. The temporal logic is then mapped to assertions to check if a safety property holds. This method not only ensures the system reliability at every stage of synthesis but also makes the debugging quicker.

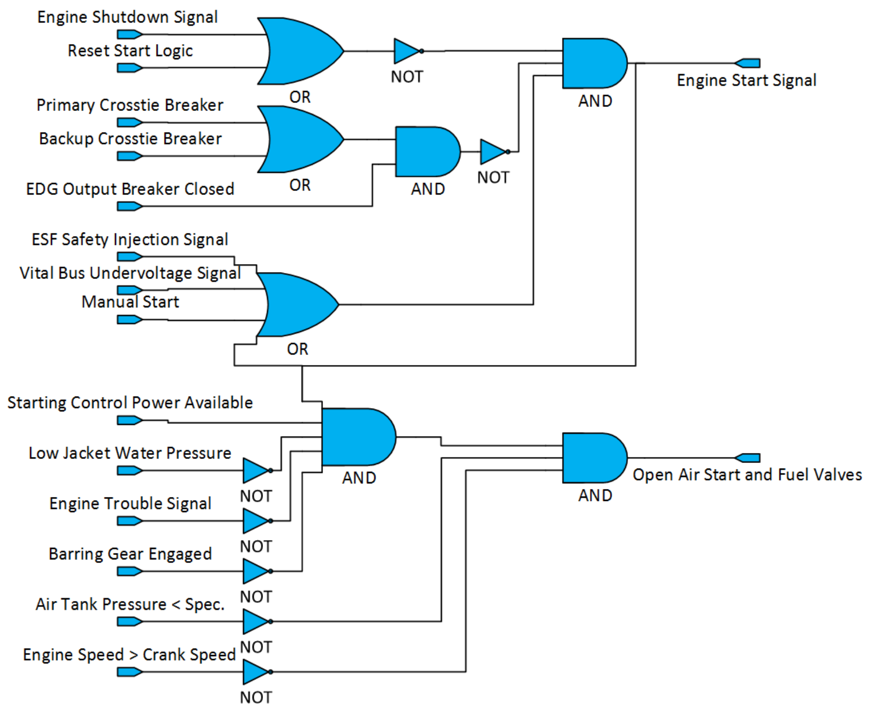
Formal verification using Questasim was realized by inserting assert statements in the RTL code generated by Simulink HDL Verifier. The assert statement specifically tests for critical conditions which are most important for the correct functionality of the system. Run time verification of the system can be achieved using hardware monitors synthesized from the assertion properties. Such runtime monitors provide strong evidence of safety assurance for the application.

5 Results

In this section we apply the SymPLe architecture and generate results for two typical applications found in a Nuclear Power Plant. The first application is the control logic for starting an Emergency Diesel Generator. The other application is a typical PID controller for controlling a process to its given setpoint. Both applications have low requirements regarding processing and computational capabilities but are good representations for target applications for SymPLe. The given results are simulations of the model-based representation of SymPLe, as the effort for implementation in an actual FPGA device is ongoing.

5.1 Emergency Diesel Generator control logic

The Emergency Diesel Generator infrastructure within a NPP is a highly critical system required for reactor cooling, other safety functionalities, and to prevent a station blackout. The functional logic for starting the EDG is shown in Fig. 9 and was published in an EPRI technical update [15]. The logic can be split into two separate portions where the first portion controls an engine start signal (upper half of the logic diagram) and the second portion controls an air and fuel valve.



Engine Shutdown Signal
Reset Start Logic
OR
NOT
AND
Engine Start Signal
Primary Crosstie Breaker
Backup Crosstie Breaker
OR
AND
NOT
EDG Output Breaker Closed
ESF Safety Injection Signal
Vital Bus Undervoltage Signal
Manual Start
OR
Starting Control Power Available
Low Jacket Water Pressure
AND
AND
Open Air Start and Fuel Valves
NOT
Engine Trouble Signal
NOT
Barring Gear Engaged
NOT
Air Tank Pressure < Spec.
NOT
Engine Speed > Crank Speed
NOT

Figure 9: Logic diagram for starting the EDG

.

The different parts of the logic are placed in separate tasks and are computed parallel in SymPLe. Running the EDG logic in the SymPLe environment generates the results in Table 2. The two categories for the performance metrics include the full deterministic execution time, always constant, and the worst case response time, which is defined as the worst time between a change of input(s) and an output response, for the EDG application. The scan-cycle time S for the EDG application is 211 CC's and is dominated by task 2 using 8 function blocks for computing the lower half of the EDG logic. The scan-cycle time can be calculated using equation 1, where $max\{s_n\} = 8$ and $d = 17$.

Table 2: EDG performance in Clock Cycles

Metric	**Clock Cycles**
Execution Time (task 1)	93
Execution Time (task 2)	106
WCRT Engine Start	387
WCRT Open Air/Fuel Valve	591

5.2 Standard PID controller

A second example of a perfect application for the SymPLe architecture is a conventional PID controller. Besides showing the logical operation of the EDG, the PI controller implementation will test several of the more mathematical intensive function blocks. Figure 10 shows the output of a plant controlled by a PID controller fully implemented in SymPLe after giving it a step input.

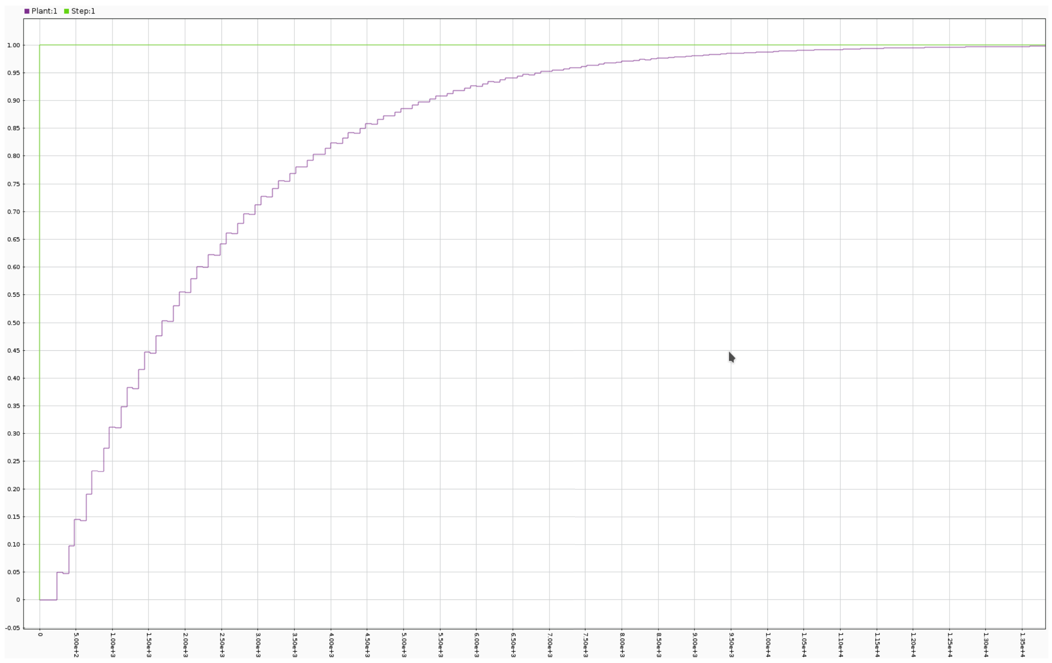


Figure 10: Running the SymPLe PID controller on a plant with a step input
.

The PID controller takes 121 CC's to yield an updated control signal for the plant process, which is equal to the scan-cycle time with 8 function blocks being sequenced and 2 input/output operations steps. It should be mentioned that it is assumed that the CC's is equivalent to the Simulation steps in Simulink but this will have to be further investigated when transitioning to an actual FPGA device.will

6 Conclusion and Future Work

SymPLe is a novel approach to a digital I&C architecture, implemented on an FPGA or ASIC, suitable for safety critical applications. SymPLe eases the integration of digital technology in systems with stringent safety requirements. It reduces common cause failures through multiple layers and boundaries before systems reach safety-critical failure.

Other current related work not written from perspective of complete design around ease in verification and validation efforts. The SymPLe architecture is designed on 3 principles: simplicity, verifiability, and determinism. It is designed in mind of adhering to stringent government requirements such as IEC-61131 and IEC-61499.

SymPLe is formally verified on all levels. The SymPLe architecture is checked at the model level using Simulink Design Verifier to ensure models perform in accordance with mathematically specified properties. At the hardware description language level, SymPLe is again re-verified using Mentor Graphics Questa model checker. Connection of the two verification tools, Simulink Design Verifier and Questa, provides end-to-end verification and validation coverage.

SymPLe has been simulated in applications typical in a Nuclear Power plant. The two applications consist of the startup logic for an Emergency Diesel Generator and for a PID controller on a modeled plant. Architecture performance metrics such as execution times and worst case response time were produced with these applications. These applications demonstrate the complete deterministic behavior of SymPLe.

Future work on the project focuses on bringing the project to maturity. Firstly, we intend to implement the current SymPLe architecture on an FPGA and begin its thorough testing rigors. Simultaneously we are investigating the leveraging of the cross-platform capabilities of SymPLe by look at placing the architecture on an ASIC. Another short term goal set for the project is to achieve the end-to-end verification previously mentioned in this article. We are also researching the potential capabilities that special function plugins or tasks could add to the architecture and the subsequent flexibility that could be added to a fully verified I&C system: SymPLe.

References

[1] U. N. R. Commission and others, "NUREG/CR 6303," *Method for Performing Diversity and Defense-In-Depth Analyses of Reactor Protection Systems, Washington, DC*, 1994.

[2] C. Plessl and M. Platzner, "Virtualization of Hardware-Introduction and Survey." in *ERSA*, 2004, pp. 63–69. [Online]. Available: http://www.tik.ee.ethz.ch/file/9647b24fcc560908e40a98f49bbcd5c4/plessl04_ersa.pdf

[3] D. Koch, C. Beckhoff, and G. G. F. Lemieux, "An efficient FPGA overlay for portable custom instruction set extensions," in *2013 23rd International Conference on Field programmable Logic and Applications*, Sep. 2013, pp. 1–8.

[4] Z. Hajduk, J. Sadolewski, and B. Trybus, "FPGA-based execution platform for IEC 61131-3 control software," *Przegld Elektrotechniczny*, vol. 87, no. 8, pp. 187–191, 2011, bibtex: hajduk_fpgabased_2011.

[5] M. Chmiel, J. Kulisz, R. Czerwinski, A. Krzyzyk, M. Rosol, and P. Smolarek, "An IEC 61131-3-based PLC implemented by means of an FPGA," *Microprocessors and Microsystems*, 2015, bibtex: chmiel_iec_2015. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0141933115001659

[6] S. Rudrawar, M. Patil, and others, "Design of Instruction List (IL) Processor for Process Control," *International Journal of Science & Emerging Technologies*, vol. 5, no. 1, 2013, bibtex: rudrawar_design_2013.

[7] A. Milik and E. Hrynkiewicz, "On Translation of LD, IL and SFC Given According to IEC-61131 for Hardware Synthesis of Reconfigurable Logic Controller," in *World Congress*, vol. 19, 2014, pp. 4477–4483, bibtex: milik_translation_2014.

[8] C. Economakos and G. Economakos, "FPGA implementation of PLC programs using automated high-level synthesis tools," in *IEEE International Symposium on Industrial Electronics, 2008. ISIE 2008*, Jun. 2008, pp. 1908–1913, bibtex: economakos_fpga_2008.

[9] S. Shukla, N. W. Bergmann, and J. Becker, "QUKU: a two-level reconfigurable architecture," in *IEEE Computer Society Annual Symposium on Emerging VLSI Technologies and Architectures (ISVLSI'06)*, Mar. 2006, pp. 6 pp.–.

[10] S. Shukla, N. W. Bergmann, and J. Becker, "QUKU: A Fast Run Time Reconfigurable Platform for Image Edge Detection," in *Reconfigurable Computing: Architectures and Applications*. Springer Berlin Heidelberg, Mar. 2006, pp. 93–98, dOI: 10.1007/11802839_13. [Online]. Available: http://link.springer.com/chapter/10.1007/11802839_13

[11] S. Shukla, N. W. Bergmann, and J. Becker, "QUKU: A coarse grained paradigm for FPGAs," in *Dagstuhl Seminar Proceedings*. Schloss Dagstuhl-Leibniz-Zentrum fr Informatik, 2006. [Online]. Available: http://drops.dagstuhl.de/opus/volltexte/2006/742/

[12] S. Shukla, N. W. Bergmann, and J. Becker, "QUKU: A FPGA based flexible coarse grain architecture design paradigm using process networks," in *2007 IEEE International Parallel and Distributed Processing Symposium*. IEEE, 2007, pp. 1–7. [Online]. Available: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=4228110

[13] K. D. Pham, A. K. Jain, J. Cui, S. A. Fahmy, and D. L. Maskell, "Microkernel hypervisor for a hybrid ARM-FPGA platform," in *2013 IEEE 24th International Conference on Application-Specific Systems, Architectures and Processors*, Jun. 2013, pp. 219–226.

[14] R. Wieringa, *Design Methods for Reactive Systems: Yourdan, Statemate, and the UML.* Morgan Kaufmann, 2003, google-Books-ID: zLvPfFbH5y4C.

[15] EPRI, "Emergency Diesel Generator Digital Control System Upgrade Requirements." [Online]. Available: http://www.epri.com/abstracts/Pages/ProductAbstract.aspx?ProductId=000000003002002098