Migration From PLC to IEC 61499 Using Semantic Web Technologies

Wenbin (William) Dai, *Member, IEEE,* Victor N. Dubinin, and Valeriy Vyatkin, *Senior Member, IEEE*

***Abstract*—This paper proposes a new methodology of migration from IEC 61131-3 PLCs to IEC 61499 function blocks. The aim of this migration process is to recreate IEC 61131-3 applications in IEC 61499 implementations with equivalent execution behavior. The formal model of the IEC 61131-3 standard for migration and cyclical execution model is defined. This method also creates a foundation for correct-by-design development tools and automatic migration between the IEC 61131-3 and IEC 61499 standard. Formal migration rules based on ontology mappings, restoring execution model including tasks and programs scheduling and variables mapping with different access levels, are provided. A transformation engine for importing PLC code, mapping from PLC ontology model to function block model and code generation is implemented based on the ontological knowledge base and semantic query-enhanced web rule language. The migration approach is demonstrated on a simple airport baggage handling system.**

***Index Terms*—Code generation and IEC 61131-3 formal execution model, execution semantics, IEC 61131-3 PLC, IEC 61499 function blocks, migration, ontology (OWL), ontology mapping.**

I. Introduction

THE IEC 61499 standard [1] is considered the key of enabling distributed control and introducing intelligence into industrial automation [2]. However, the use of this technology in the automation industry is still minimal. A majority of the systems are still designed using programmable logic controllers (PLCs) programmed in traditional languages of ladder logic, structured text, or function block diagram (FBD) defined in the IEC 61131-3 standard [3]. Potential benefits of implementing complex automation systems with IEC 61499 technology include lower design effort combined with higher flexibility, reconfigurability, and maintainability [4]–[7], but the learning curve is quite steep and the cost of required initial research and development is high. Therefore, it is important to provide an easy migration path for existing PLC programs into IEC 61499 compliant platforms as the first step toward widespread adoption of the new standard.

There exist several migration approaches [8]–[12] but none of them provides a generic rule for automatic generation an IEC 61499 system from code developed for various PLC brands. Although most vendors claim their PLCs are compliant with the IEC 61131-3 standard, none of them is compatible with each other [13]. The migration process proposed in this paper is not trying to replace the IEC 61131-3 standard with the IEC 61499 standard completely. This complies with the vision of many industry practitioners (see [20]) who see IEC 61499 as a complement rather than a substitute to IEC 61131-3. Instead, the IEC 61499 standard is used as the top-level design framework in conjunction with the IEC 61131-3 standard. In this approach, entire IEC 61131-3 PLC programs are encapsulated in the IEC 61499 function blocks and reused in the new IEC 61499 system configuration. In order to achieve cyclic execution behavior of the original PLC applications in the generated IEC 61499 systems, a formal IEC 61131-3 execution model is defined. However, the first edition of IEC 61499 allowed for some semantic ambiguities, leading to several execution models for IEC 61499: sequential [21], parallel [22], cyclic [23], and synchronous [24] (in more detail discussed in Section IV). Due to the different execution semantics, same IEC 61499 configurations could have different behaviors when run on various execution semantics. In order to avoid that, the equivalent execution model in IEC 61499 is built on application level according to the PLC execution model.

This paper proposes a novel, tool-supported transformation process implemented by a tool based on the semantic web technologies [43]. The proposed ontology-based approach is using knowledge bases to describe the model and transformation between the models. This substantially differs from the usual XML transformation methods, such as the ones based on XSLT translation table or XQuery, in which case it would be much harder to take into account the execution semantics of the transformed artifacts, and translation rules would be hard coded. In the proposed method, source code of PLC programs is converted into a platform independent ontology knowledge base. The execution order of the PLC source code is also stored in the same knowledge base. Then, all instances of this IEC 61131-3 ontological model are transformed to the IEC 61499 ontological model based on the generic migration rules. The scheduling function blocks are inserted to control the execution order of the resulting IEC 61499 function blocks. Finally, the resulting IEC 61499 system configuration is generated automatically from the IEC 61499 knowledge base.

The execution behavior of the migrated IEC 61499 systems is identical to the IEC 61131-3 version, taking into account

Manuscript received July 16, 2012; revised February 5, 2013; accepted May 10, 2013. Date of publication November 13, 2013; date of current version February 12, 2014. This paper was recommended by Associate Editor T. I. Strasser.

W. Dai and V. Vytakin are with Luleå University of Technology, Luleå 971 87, Sweden (e-mail: w.dai@ieee.org; vyatkin@ieee.org).

V. Dubinin is with the University of Penza, Penza 440000, Russia (e-mail: victor_n_dubinin@yahoo.com).

Color versions of one or more of the figures in this paper are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/TSMCC.2013.2264671

tasks priorities and interruption between periodic and continuous tasks. However, thanks to IEC 61499 features, the migrated function block systems can be easier distributed to multiple small PLCs without any code change. Communications between FB instances are established automatically at runtime level in IEC 61499 implementations. As most of PLC code is reused in new function block systems, the cost of redesigning IEC 61131-3 systems in IEC 61499 can be reduced significantly. Using the IEC 61499 standard as a top level design language provides a better overview of systems compared to IEC 61131-3 programs. Finally, the insertion of scheduling function blocks during the migration process ensures that the generated systems behave identically to their PLC version even if different distributed nodes follow different execution semantics of IEC 61499.

The rest of the paper is structured as follows: several migration and transformation approaches are reviewed in Section II. In Section III, the overview of the migration path from IEC 61131-3 to IEC 61499 using ontological knowledge base is illustrated. In Section IV, the execution model of the IEC 61131-3 is defined and an equivalent IEC 61499 version is created. Importing code and structure from PLC programs are discussed in Section V. In Section VI, general migration rules are provided as ontology mapping between the knowledge bases. Code generation for IEC 61499 is provided in Section VII, followed by a case study of an airport baggage handling system (BHS). The paper is concluded with discussion of limitations and future works.

II. Related Works

Several researchers have published on migration from IEC 61131-3 PLCs to IEC 61499 FBs. Peltola *et al.* [8] provided an example on manual migration of IEC 61131-3 PLC control code for a batch process to IEC 61499 function blocks with integrated HMI views. The paper has proven that using IEC 61499 function blocks replacing for IEC 61131-3 PLCs is feasible.

Another example of manual PLC controller migration to an IEC 61499 distributed control system is given by Hussain *et al.* [9]. PLC-based control of a FESTO MPS model assembly plant is transformed to the IEC 61499 function block-based control. The original control algorithm in state machine is converted to the execution control chart (ECC) of the basic function block (BFB). The performance of the new FB system based on event-driven execution is measured. However, no generic rule or migration guideline is provided.

A model-based transformation and semantic correction from IEC 61131-3 to IEC 61499 is proposed by Wenger *et al.* [10], [11]. Authors proved that with IEC 61131-3 and IEC 61499 libraries available, it is possible to transform source IEC 61131-3 model into an internal E-core model and then write it in the target IEC 61499 XML model. Recreating IEC 61131-3 execution order in IEC 61499 is feasible but how to connect events in order automatically is still not completely solved. However, this approach focuses on the translation of the PLC code sentence by sentence. The entire PLC program structure was not considered.

Sunder *et al.* [12] proposed a solution for converting IEC 61131-3 automation projects into IEC 61499 standard. Differences between two standards are carefully compared. The mapping between two software models is also provided. The variables with various level of access are considered. However, mapping of global variables that are commonly used by IEC 61131-3 to an IEC 61499 format is not solved and automatic transformation is not implemented using this approach.

Beyond existing migration approaches, there are several papers providing useful ideas that could be applied in the migration process. Basile *et al.* [14] show how object-oriented programming could implicitly make the event-based PLC software behavior. Authors propose a solution that is the combination of two existing approaches in introducing object-oriented programming into PLCs: adding object-oriented support in the IEC 61131-3 standard or adopting the IEC 61499 standard. The IEC 61131-3 programming language sequential function chart (SFC) is used to develop a proper object-oriented approach. Our work goes beyond object-oriented programming, focusing on migration to distributed systems, which is the main strength of the IEC 61499 architecture.

Goh and Dint [15] describe an approach to code generation for IEC 61499 based on the iterative knowledge base. The iterative knowledge base is represented in the form of XML and Extended Backus-Naur Form (EBNF). The goal of that approach is to eliminate any additional script language to be used in the code generation. Also the translation rules are configurable and reusable to improve the accuracy of translation rules. In order to achieve this goal, rule-based blocks are built for each IEC 61499 XML element. During the code generation process when the predefined rules are satisfied, the related block of code will be generated and data types and connections will be also inserted. XML and EBNF-based approach is syntax-oriented while OWL, semantic web rule language (SQWRL)/SWRL-based ontology driven approach is more semantic-oriented and human readable. Besides, automatic generation of code template is not covered in that paper. It could be difficult to modify knowledge base when code template is changed.

Younis *et al.* [16] propose an XML and XSLT-based approach for reimplementing existing PLC programs. State machines stored in XML format are translated into IEC 61131-3 POUs and regenerated for a different controller. This work only covers migration between IEC 61131-3 platforms and original PLC code must be defined using assembler-like languages.

Weisenborn *et al.* [17] registered a patent on conversion from a hardware configuration and the corresponding control logic program from one PLC to another equivalent PLC using a knowledge-based translator. However, the premise of this approach is that the hardware configurations must be equivalent. The migration process described in the following section aims at transformation between multiple platforms supporting various PLC programming languages, while the approach of [17] uses a proprietary language for describing mapping rules and is intended mainly for migration of ladder logic control programs.

Fay *et al.* [18] propose an automatic graphical knowledge-based approach to address the modernization problem of a control system for a technical plant. This approach proves rule-based transformation is very useful for function block-based control program. However, the platform used in this approach is not compliant with the IEC 61499 standard. In addition, this approach needs to develop a specific rule execution engine. Its formats are proprietary that complicates integration of the proposed tool with other tools of the design chain. On the contrary, semantic web-based approaches support open standards, expansibility, reconfigurability, reuse of previous decisions, and use standard software (for example, reasoners).

Mader *et al.* [19] investigate how to obtain proper functioning from PLC applications using formal methods. A schema of PLC applications is presented in this paper to indicate how to evaluate results gained by formal methods as well as find out which aspects are treated by a certain method in the control system design. Unfortunately, the schema proposed describes the process of PLC-based systems design through analysis but it does not show the role of formal methods in the process of synthesis of these systems.

Overall, there is no systematic approach to automatic migration between an IEC 61131-3 PLC and IEC 61499 function blocks available. Approaches from [8] and [9] provide some useful migration ideas, but formulating migration rules is missing. In this paper, such migration rules are proposed. Most of the rules proposed by Sunder's approach can be applied here. Mapping of global variables, missing in the Sunder *et al.* work, is also covered in this paper. The transformation process proposed by Wenger *et al.* is driven by a meta-model defined in the standard XML format. The approach does not provide the mapping between other IEC 61131-3 languages except FBD to IEC 61499 as well as is not generic which could be applied to other IEC 61131-3 XML formats. The mapped function block systems using this approach cannot automatically connect event connections and associate them with proper data variables. Furthermore, none of the existing approaches offers an identical IEC 61131-3 execution model in IEC 61499. This paper aims at addressing these issues.

III. Migration Process from IEC 61131-3 to IEC 61499 using Ontology

The proposed migration approach aims at automatic recreation of IEC 61131-3 PLC code behavior in IEC 61499. The entire process includes importing of PLC programs into a knowledge base, automatic mapping from IEC 61131-3 knowledge-base elements to IEC 61499 version, recreation of original execution models, and code generation to IEC 61499 platforms. The proposed design approach is illustrated in Fig. 1.

On the left-hand side of Fig.1, source code of single PLCs written in IEC 61131-3 languages is imported into the IEC 61131-3 knowledge base as instances. The IEC 61131-3 knowledge base is defined in the semantic web language OWL [25]. An ontological knowledge base consists of two parts: T-Box and A-Box. In the ontological terms, definitions are belonging to the taxonomy box (T-Box) that contains general

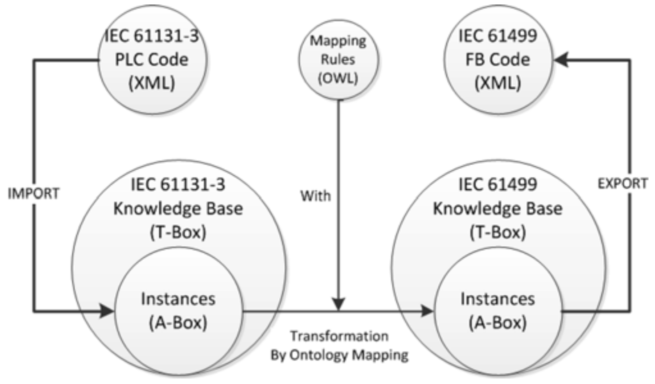


Fig. 1. Migration process from IEC 61131-3 PLCs to IEC 61499 FBs.

properties of the ontological concepts. When this is applied to the IEC 61131-3 standard, a T-Box contains definitions of all IEC 61131-3 program elements concepts in the XML format and program hierarchy between those IEC 61131-3 concepts. Those concepts are linked together by using object properties and described in the description logic [26] which ontology is mainly defined in. Actual data of concepts are stored in data properties. In the other part of the knowledge base, instances are stored into the assertion box (A-Box) which consists of knowledge that is specific to the individual system design. In the migration process, an A-Box retains all instances imported from PLC configurations. That information is saved as ontology individuals in the A-Box.

The source code of PLCs represented in XML is interpreted during the import process and stored into the IEC 61131-3 knowledge base in the OWL format which relies on XML as well. The import process consists of two parts. The first part is to detect PLC execution order including scheduled parameters of tasks and programs. This step also gathers PLC tasks execution order, priority and other useful information that is required later in the migration process. The second part is to store the original algorithms encapsulated in function blocks, functions, and subroutines written in IEC 61131-3 languages. Those algorithms are considered as fundamental components and will be reused again. If algorithms in IEC 61131-3 function blocks with protected source code still can be used in the migrated code being encapsulated into event-driven IEC 61499 function block. Details of such encapsulation are subject to support at the device level. For example, such FBs can be called from algorithms in basic FBs.

After the knowledge base for the original PLC code has been created, the next action is to transform all IEC 61131-3 elements into the IEC 61499 format according to the general migration rules. The execution model of IEC 61131-3 is recreated by inserting scheduling function blocks in IEC 61499. Tasks and programs from PLC codes are rearranged according to the original order with new introduced scheduling function blocks. The transformation engine is based on the semantic query-enhanced web rule language (SQWRL) [27] that is intended to extract information from OWL knowledge bases. Similar to SPARQL, which queries RDF knowledge base [28], SQWRL is the query language for OWL. Mapping rules defined as complete SQWRL queries are taken into the transformation engine. The results of those queries provide the information required to construct desired IEC 61499 systems.

The last step in the chain is the code generation part. Based on the target platform, the respected code templates and related instances are combined by the code generation engine and output as IEC 61499 XML files. The generated code can be viewed and edited in any IEC 61499 IDE and immediately deployed using its mechanisms of compilation and deployment. The example platform is this paper is FBDK [29] although there are many IEC 61499 platforms such as nxtStudio [30], 4DIAC-IDE [31], FBench [32], and CORFU [33].

Further lifecycle steps, such as deployment to distributed devices and modifications are performed using IEC 61499 tools. One major advantage of the IEC 61499 standard is facilitation of distributed control by providing flexibility for code distribution. The generated IEC 61499 systems can be deployed to multiple distributed controllers by simply assigning target device for each FB instance. The event-driven IEC 61499 execution and communication model is based on message passing which guarantees correctness of execution regardless of performance of devices and networks. A PLC-based distributed system must use shared variables concept, whose semantics will depend on timing of each PLC scan and of the communication networks involved. Besides, IEC 61499 engineering tools automatically insert communication code when connections between function blocks cross boundaries of devices. On the other hand, implementing data sharing by several PLCs is a lot more cumbersome requiring lots of manual modifications. Also using global variables in PLC program could reduce the system reliability and performance and increases maintenance costs in general.

Another advantage is that the IEC 61499 approach promotes component-based design [34]. Functions or function blocks designed in original PLC programs are transformed by the migration process to reuseable library components. Those components could be reused over and over again. Maintenance of the resulting migrated code with IEC 61499 tools will not be difficult due to the use of the same mnemonic notation of variable names and code labels (e.g., states and actions as shown in Fig. 15 below). Original function names are identical to target function block names as well as instance names as illustrated in Figs. 19 and 20 in the case study. Minor specific changes that are not supported by the migration process could be added by using IEC 61499 tools. Major changes could be achieved by modifying source IEC 61131-3 programs and regenerate target IEC 61499 programs again. The migration process provides a momentary result in some cases (distributed PLC systems). The main effort of the migration process is to help at the initial stage of a transition from centralized PLC-based control to distributed control systems.

IV. Formal Models of IEC 61131-3 for Migration

Before starting migration process, one more important issue needs to be resolved, namely, the execution semantics. In the migration approach proposed in this paper, the execution semantics of PLC is recreated by means of IEC 61499. This includes not only cyclic execution model which is achievable

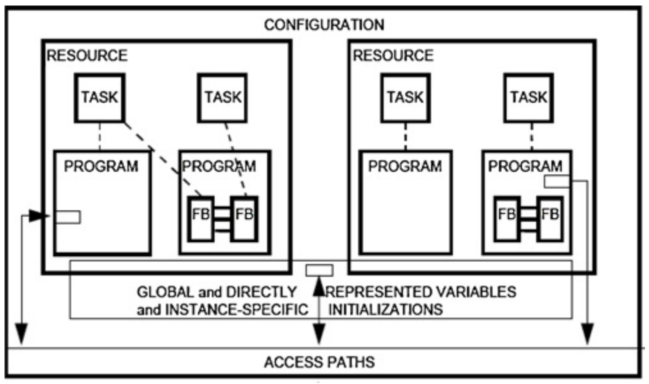


Fig. 2. IEC 61131-3 software model [3].

by some function block runtimes (e.g., ISaGRAF [35]), but also other PLC key features such as priority of tasks and interrupts between periodic and continuously executed tasks.

As a starting point, a formal model of IEC 61131-3 is required. The IEC 61131-3 basic high level language elements and their relationships are illustrated in Fig. 2.

In an IEC 61131-3 system configuration, each resource corresponds to a physical device—PLC. Every PLC may have one or more tasks which contains one or more programs. Inside each program, functions and function blocks can be invoked from any program of any task in the same resource. Programs, functions, and function blocks are fundamental programming organization units (POUs) of IEC 61131-3. Nested structures of POUs are also supported. Four programming languages are defined in IEC 61131-3. Graphical languages include ladder logic diagram (LD) and function block diagram (FBD). Structure text (ST) and instruction list (IL) are textual-based languages. All languages can be encapsulated into POUs. SFC is defined in common elements of the IEC 61131-3 standard to represent logic in the form of state machines.

For the purposes of formal definition of transformations, an IEC 61131-3 resource is defined as a tuple

$$Res = (Task, GlobalVar, UseTG, UpdTG) \tag{1}$$

where $Task = \{task_1, task_2, \ldots, task_n\}$ is a nonempty set of tasks scheduled in the PLC; $GlobalVar = \{globalVar_1, globalVar_2, \ldots, globalVar_m\}$ is a set of global variables of the resource; $UseTG \subseteq Task \times GlobalVar$ is a relation of using the global variables in the tasks (for reading); $UpdTG \subseteq Task \times GlobalVar$ is a relation of changing the global variables in the tasks.

Each task is defined as a tuple

$$task = (Program, LocalVar, UsePL, UpdPL) \tag{2}$$

where $Program = \{program_1, program_2, \ldots, program_p\}$ is a nonempty set of programs scheduled in a PLC task; $LocalVar = \{localVar_1, localVar_2, \ldots, localVar_s\}$ is a set of local variables used in one or more PLC program of this task; $UsePL \subseteq Program \times LocalVar$ is a relation of using the local variables in the programs (for reading); $UpdPL \subseteq Program \times LocalVar$ is a relation of changing the local variables in the programs. There must be some variables defined in the resource so:

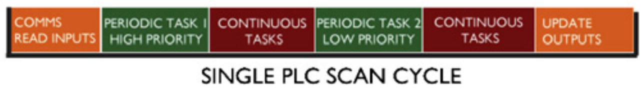


Fig. 3. IEC 61131-3 execution semantics timing diagram.

For $task_i$ and $task_j$ $(i \neq j)$ it holds

$$Program^i \cap Program^j = \emptyset.$$

Here the upper indexes i and j are used to refer to the corresponding tasks.

Function blocks can be called along with algorithms inside a program

$$Program = (Alg, FB, Func) \tag{3}$$

where Alg is representing the actual PLC algorithms consists of statements and operators written in one of the five programming languages in this particular program not including hierarchy levels of functions or function blocks invoked; $FB = \{FB_1, FB_2, \ldots, FB_k\}$ is a set of function block instances used in the program; $Func = \{Func_1, Func_2, \ldots, Func_h\}$ is a set of functions instances used in the program.

A function block is defined as a 3-tuple

$$FB = (Interface, AlgFB, Var) \tag{4}$$

where $Var = \{var_1, var_2, \ldots, var_x\}$ is a set of internal variables used in the function block; *AlgFB* is an algorithm incorporated into *FB*. *Interface* in *FB* is defined as

$$Interface = (VI, VO, VIO, EN, ENO) \tag{5}$$

where $VI = \{vi_1, vi_2, \ldots, vi_r\}$ is a set of input variables; $VO = \{vo_1, vo_2, \ldots, vo_q\}$ is a set of output variables; $VIO = \{vio_1, vio_2, \ldots, vio_t\}$ is a set of in/out variables; *EN* is a Boolean-type input variable to indicate the function block activation status; *ENO* is a Boolean-type output variable to indicate the function block is operating.

After the basic elements of IEC 61131-3 are defined, the execution function using those elements is also required for the migration process. The IEC 61131-3 execution is based on scan cycles [3]. As shown in Fig. 3, during each scan, a PLC runs communication services first, reads all inputs data from input modules, then executes through all tasks and updates all outputs data to the out modules finally. The execution order in this model is sequential. A new execution cycle starts immediately once the previous cycle is completed.

The formal IEC 61131-3 execution model for a PLC resource is given as a tuple

$$M = (Res, If, \{Tf_i\}_{i=\overline{1,n}}, Of) \tag{6}$$

where *If* is a function servicing input data from the input PLC modules; *Of* is a function updating data to the output PLC modules; Tf_i is a task execution function $(i = \overline{1, n})$ which consists of multiple program execution functions $Pf^i_j (j = \overline{1, p^i})$; *Res* refers to an IEC 61131-3 resource as defined in (4.1).

The global variables can be divided into three classes: $GlobalVar = InputVar \cup OutputVar \cup UDVar$.

$$InputVar \cap OutputVar \cap = \emptyset$$
$$GlobalVar \bigcup \bigcup_{i=1}^{n} LocalVar^i \neq \emptyset$$

where *InputVar* is a set of variables mapping from the PLC input modules; *OutputVar* is a set of variables mapping to the PLC output modules; *UDVar* is a set of variables defined by the users to be used in multiple programs.

The essence of function *If* is to sample the values of PLC data inputs to variable from *InputVar*. Function *Of* "copies" the values of variables from *OutputVar* to PLC data outputs.

There are two types of tasks in the IEC 61131-3 standard: periodic tasks and continuous tasks. Periodic tasks execute at a customizable nonzero time interval. Continuous tasks keep looping through all associated programs until interrupted by periodic tasks. Each task also has a different preset priority level. In a preemptive PLC runtime environment, a lower priority periodic task will be interrupted by a higher priority periodic task when the execution of this lower priority task cannot be accomplished before the next turn of that higher priority task starts. Once the higher priority task terminates, the lower priority task will resume execution at the place where it was interrupted. In a nonpreemptive PLC runtime environment, the higher priority task will wait until the lower priority task terminates. If two tasks have same priority and all waiting for execution, the task with longer waiting time will have the priority. For the continuous tasks, the priority means the execution order. Higher priority continuous tasks always execute first followed by the lower priority continuous tasks. The execution of the continuous task will resume from the place where it was interrupted as well. Given an IEC 61131-3 resource *Resource* as defined in (4.1), the task execution function for $task_i \in Task$ is defined as $Tf_i = \bigcup_j Pf^i_j$.

The execution function Pf^i_j of the $program_j$ in the $task_i$ is defined as

$$Pf^i_j : \prod_{(task_i,\, globalVar_k) \in UseTG} Dom(globalVar_k) \times \prod_{(program_j, localVar_t) \in UsePL^i} Dom(localVar_t) \rightarrow$$

$$\prod_{(task_i, globalVar_i) \in UpdTG} Dom(globalVar_z) \times \prod_{(program_j, localVar_u) \in UpdPL^i} Dom(localVar_u)$$

The algorithm for a PLC execution scan is given as

```
1: for all $task_i \in$ Continuous Task do
2: for all $program_j \in Program^i$ do
3: if $task_i$ is interrupted by $task_k \in$ Periodic Task then
4: for all $program_m \in Program^k$ do
5: Compute $Pf^k_m$
6: end for
7: end if
8: Compute $Pf^i_j$
9: end for
10:end for
```

The scan time of a PLC cycle time is defined as

$$Tscan = \sum_i (x_i \times TP_i) + \sum_j TC_j + Tcomm$$

when $Tcomm$ is also the time of servicing communications and I/Os; TP_i is ith individual periodic task execution time; TC_j is jth individual continuous task execution time; x_i is representing the number of times periodic tasks execute.

In a PLC cycle, all scheduled continuous tasks will be executed once. The periodic tasks will execute zero or more times depends on duration of each PLC scan cycle time. If the scheduled continuous tasks execute fast enough, the periodic tasks may only be executed every several scan cycles. Or if the scheduled continuous tasks execute extremely slow, the periodic tasks may execute several times before the scan terminates.

In a task, there are multiple programs executed in a sequential sequence. The total execution time of a task is

$$Ttask = \sum_i Tpro_i$$

where $Tpro_i$ is ith individual program execution time.

On the other hand, the IEC 61499 standard has a variety of execution semantics as stated in the Section I, three of which are most known: sequential, parallel, and synchronous. In the sequential semantics, all FB instances run like in a single thread with only FB active at any moment of time. In the parallel execution semantics several FB instances can be active simultaneously. In the synchronous execution semantics, the concept of logical tick is introduced as an interval in which all FBs are executed with the same set of input data. Their exchange events and data at the end of one tick, and take the changes into account in the next tick. In the cyclic model, all FBs are executed once in a predefined sequence, between updates of input and output variables. Synchronous and cyclic execution semantics are similar in a way to cyclic scan based execution of programs in PLCs.

In order to apply the IEC 61131-3 execution model in all those runtime semantics, a scheduler system in function blocks is built at the application level. The hierarchy of this proposed scheduler system given in Fig. 4 ensures PLC cyclical behavior—only one function block of a program in a task is activated at one time. There are three layers of schedulers: a PLC main scheduler, a task scheduler, and a program scheduler.

The PLC main scheduler controls the order of tasks' execution. The main scheduler is a BFB type and its interface, ECC and algorithms are illustrated in Fig. 5 for a preemptive PLC scan.

There are three tasks in this example. P1 and P2 are periodic tasks with different scan time and priority. In this case, P1 has a faster scan time of 20 milliseconds (ms) and higher priority than P2 which executes every 35 ms. C1 refers to a continuous task. Here C1 is preempted by P1 and P2.

The PLC main scheduler is implemented as a BFB with the following structure. Its ECC includes four states: INIT, REQ, P1_DONE, and P2_DONE. The core part REQ state is

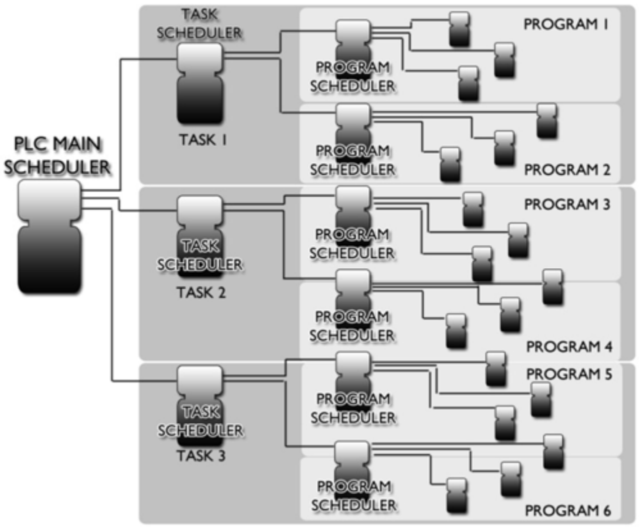
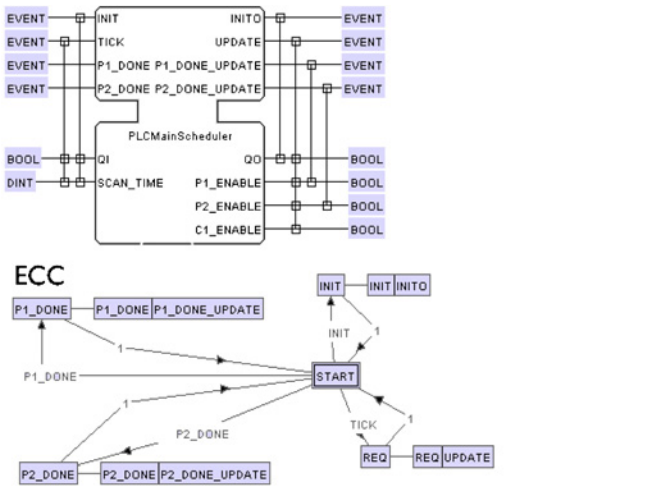


Fig. 4. IEC 61499 scheduler system for IEC 61131-3 execution semantics.



```
REQ ALGORITHM
TIMESTAMP := TIMESTAMP + 1;
IF TIMESTAMP >= SCAN_TIME THEN
        TIMESTAMP := TIMESTAMP - SCAN_TIME;
        P1_ENABLE_ONS := FALSE;
        P2_ENABLE_ONS := FALSE;
END_IF;
IF TIMESTAMP >= 20 & NOT(P1_ENABLE)
        & NOT(P1_ENABLE_ONS) THEN
        P1_ENABLE := TRUE;
        P1_ENABLE_ONS := TRUE;
END_IF;
IF TIMESTAMP >= 35 & NOT(P1_ENABLE) & NOT(P2_ENABLE)
        & NOT(P2_ENABLE_ONS) THEN
        P2_ENABLE := TRUE;
        P2_ENABLE_ONS := TRUE;
END_IF;
IF NOT(P1_ENABLE) & NOT(P2_ENABLE) THEN
        C1_ENABLE := TRUE;
ELSE
        C1_ENABLE := FALSE;
END_IF;
```

```
P1_DONE ALGORITHM
P1_ENABLE := FALSE;

P2_DONE ALGORITHM
P2_ENABLE := FALSE;

INIT ALGORITHM
TIMESTAMP := 0;
P1_ENABLE_ONS := FALSE;
P2_ENABLE_ONS := FALSE;
P1_ENABLE := FALSE;
P2_ENABLE := FALSE;
```

Fig. 5. Main scheduler interface, ECC, and scheduling algorithms (for preemptive PLC execution semantics).

triggered by the TICK event that is raised by an E_CYCLE service interface function block (SIFB) (say, every millisecond in FBDK, this can be shorter if microsecond is supported for E_CYCLE in other platforms). An internal timestamp is counted up by one every millisecond as well. This timestamp is cleared in the INIT state once the system started. It is continuously counting up to the preset PLC scan time value from

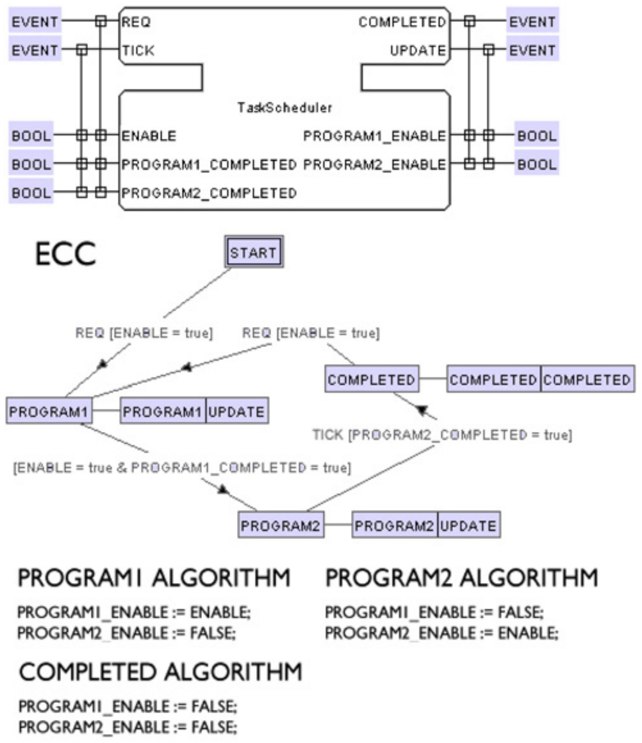
SCAN_TIME input then wrapping around back to zero again. There is a dedicated enable signal output for each task in the main scheduler. When a task is scheduled to be executed, the corresponding enable bit is set. For example, when timestamp is counted up to 20 ms, the task P1 will be activated and P1_ENABLE is set to true. The P1 will be enabled unconditionally due to its highest priority in the entire PLC configuration and preemptive PLC scan. For a nonpreemptive PLC scan, P1 will only be enabled when no other periodic task is enabled. A terminate event will be emitted from the P1 task scheduler when it ends processing. This P1_DONE event will trigger the P1_DONE state which resets P1_ENABLE signal back to false. Another internal Boolean variable P1_ENABLE_ONS is set once the P1 task is triggered every scan. It is reset by the end of the scan to ensure no periodic task is triggered more than once in single scan. Similar task trigger and task termination feedback mechanisms are set for P2 as well. When the timestamp is counted up to 35 ms, it will activate the P2 task scheduler by setting the P2_ENABLE signal to true. However, the P2 execution will be held until the P1 finishes processing due to its lower priority. The continuous task C1 will be enabled as long as no periodic task is enabled.

On the next level of the scheduling system, the task scheduler is used for scheduling the execution order of associated programs. The standard interface, ECC, and algorithms of the task scheduler function block are provided in Fig. 6.

When the main scheduler enables a task, an event at the REQ input is received with ENABLE signal set to true. It will immediately activate the PROGRAM1 state and PROGRAM1_ENABLE signal is raised to trigger the PROGRAM1 program scheduler. Once the PROGRAM1 execution is completed and this task is still enabled, PROGRAM2 will start execution without any delay. Same as the main scheduler, the TICK event input is connected to the one millisecond E_CYCLE SIFB. If this task is interrupted by another higher priority task during processing the PROGRAM1, the ENABLE bit will be reset back to false. The task scheduler will complete the PROGRAM1 execution and wait at PROGRAM1 state. When this program is scheduled to be executed in the next scan cycle, the ENABLE bit will be set again and the task will resume from executing the PROGRAM2. Once all programs in the chain are processed, a done signal is sent back to the main scheduler to switch to the next scheduled task. The task scheduler will remain idle until be activated again in the next round.

The program scheduler is responsible for arranging the execution order of routines, functions, and function blocks inside a particular program. The program scheduler function block is very similar to the task scheduler function block. Instead of scheduling programs, the program scheduler enables routines, functions, or function blocks in the predefined order. The ECC state will be named following functions, routines, or function blocks rather than using program names in the task scheduler.

The sequential execution semantics of any IEC 61131-3 configuration can be implemented in any IEC 61499 execution semantics with this scheduling system. A similar approach has been demonstrated and validated in [36].



```
PROGRAM1 ALGORITHM
PROGRAM1_ENABLE := ENABLE;
PROGRAM2_ENABLE := FALSE;

PROGRAM2 ALGORITHM
PROGRAM1_ENABLE := FALSE;
PROGRAM2_ENABLE := ENABLE;

COMPLETED ALGORITHM
PROGRAM1_ENABLE := FALSE;
PROGRAM2_ENABLE := FALSE;
```

Fig. 6. Task scheduler interface, ECC, and algorithms.

It should be noted that in order to increase system performance and responsiveness all the schedulers can be implemented in the form of SIFB rather than as basic FB.

V. Migration Mapping by Querying the Ontological Knowledge base

A. *Knowledge Base for Migration Mapping Rules*

All scheduler function blocks are not unique for different PLC program structures. Although the function block pattern is generic, customization based on the number and the type of tasks as well as the number and the execution order of programs of each project is still required. It is essential to auto generate scheduling functions as well as all mapping other function blocks for both usability and generality purposes.

In order to create the scheduling function blocks and map other tasks and programs automatically, a set of generic migration mapping rules is purposed. These generic migration mapping rules are providing the ontology mapping of classes and properties between the IEC 61499 and IEC 61131-3 knowledge bases. The structure of mapping rules knowledge base is illustrated in Fig. 7.

As defined in the knowledge base, each migration rule is stored as a step. Each step may have one or more actions. The step ontology individual template is given in Fig. 8. All step individuals with name *Step_Name* have a type of *Step* and associated with one or more actions identified by *Action_Name*. Nonterminal elements representing names are marked in bold in the following templates.

An action is responsible for mapping from one OWL class/object property/data property in the original knowledge

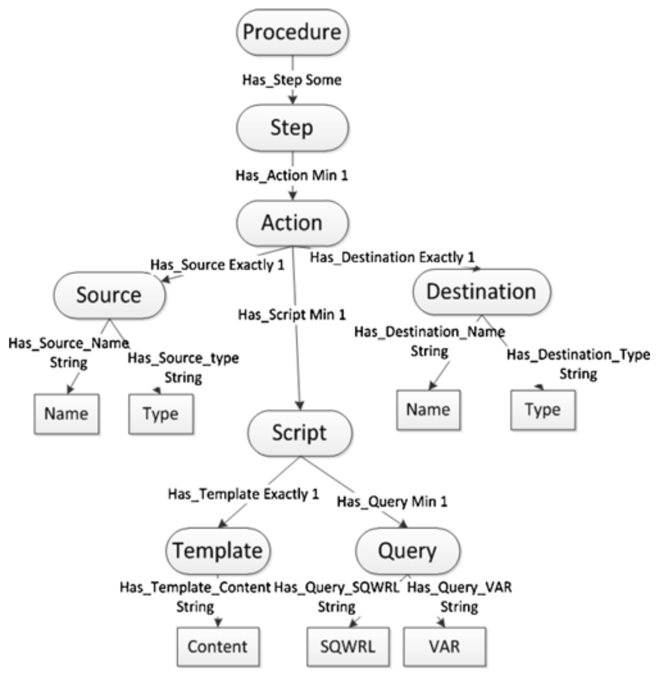


Fig. 7. Knowledge-base definition for migration mapping rules.

```
<owl:NamedIndividual rdf:about="&xsd;Step_Name">
   <rdf:type rdf:resource="#Step" />
   <Has_Action rdf:resource="#Action_Name"  />
                                          ...
</owl:NamedIndividual>
```

Fig. 8. Step ontology individual template.

```
<owl:NamedIndividual rdf:about="&xsd;ActionInstance_Action_Name">
   <rdf:type rdf:resource="#Action" />
   <Has_Source rdf:resource="#Source_Name " />
   <Has_Destination rdf:resource="#Destination_Name"/>
   <Has_Script rdf:resource="#Script_Name"  />
   <Has_Action_Name rdf:datatype="#CDATA">Action_String_Name
   </Has_Action_Name>
</owl:NamedIndividual>
```

Fig. 9. Action ontology individual template.

base to another OWL class/object property/data property in the target knowledge base. An action consists of a source, a destination, and a script. The template for the action is given in Fig. 9.

The source refers to the original node in the IEC 61131-3 knowledge base. The name *Source_Name* of the original node and the mapping type (Class/Object Property/Data Property) are associated with the source (Fig. 10).

Similar to the source, the destination is linked to the target node in the IEC 61499 knowledge base with its node name and type (Class/Object Property/Data Property). In an action, there can be one or more scripts. The script is associated with a template and a query as shown in Fig. 11.

```
<owl:NamedIndividual rdf:about="&xsd;Source_Name">
   <rdf:type rdf:resource="#Source" />
   <Has_Source_Name rdf:datatype="#CDATA">
   Original_Node_Name</Has_Source_Name>
   <Has_Source_Type rdf:datatype="#CDATA" >
   (Class/ObjectProperty/DataProperty)</Has_Source_Type>
</owl:NamedIndividual>
```

Fig. 10. Source ontology individual template.

```
<owl:NamedIndividual rdf:about="&xsd;Script_Name">
   <rdf:type rdf:resource="#Script" />
   <Has_Template rdf:resource="#Template_Name"  />
   <Has_Query rdf:resource="#Query_Name"  />
</owl:NamedIndividual>
```

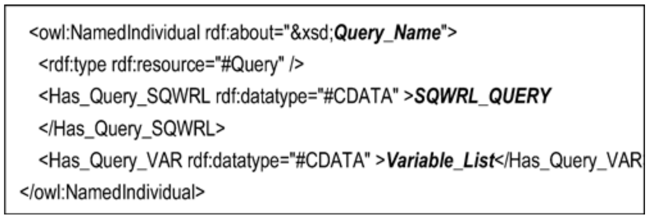
Fig. 11. Script ontology individual template.

```
  <owl:NamedIndividual rdf:about="&xsd;Template_Name">
   <rdf:type rdf:resource="#Template" />
    <Has_Template_Content rdf:datatype="#CDATA" >Content
    </Has_Template_Content>
</owl:NamedIndividual>
```

Fig. 12. Template ontology individual template.

The template in Fig. 12 keeps the content of the actual value to be written into the target node. Content is a data template that consists of a combination of text and SQWRL query target variables. The migration engine will pick up the content in the template first then replace variable symbols with the SQWRL query results for those variables. For example, the content is defined as "System_?a." ?a is a variable in the SQWRL query and VAR node for selecting the system name. The system name will be replaced in the content and finally result is shown as "System_*Name*."

Some values from the original node may be required as a part of the target value. A query engine for fetching relevant information from the ontological knowledge base would be helpful. The SQWRL provides the ability to extract information from the ontological knowledge base. SQWRL is based on the SWRL [40] and extended with string processing, aggregation, counting, and many other features. The SQWRL query (more precisely, its left-hand side) can be stored in the SQWRL node and the result variable name(s) (that is after keyword "select" in SQWRL query) is kept in the VAR node. If there is more than one result in the result set, the variable will be replaced by multiple results as one single text block. When the migration process starts, the migration engine will fetch all steps with actions from a procedure and run through them. In each action, an OWL individual instance is created if the destination type is an OWL class. This instance name is provided by the content of the template which is filled by the SQWRL query from the Query node (Fig. 13). If the destination type is a relation, an object property instance or a data property instance is created and attached to that OWL



```
<owl:NamedIndividual rdf:about="&xsd;Query_Name">
  <rdf:type rdf:resource="#Query" />
  <Has_Query_SQWRL rdf:datatype="#CDATA" >SQWRL_QUERY
  </Has_Query_SQWRL>
  <Has_Query_VAR rdf:datatype="#CDATA" >Variable_List</Has_Query_VAR
</owl:NamedIndividual>
```

Fig. 13. Query ontology individual template.

individual instance. The referred OWL individual instance from the object property instance or the value from the data property instance is selected by the query in the SQWRL node.

A migration engine is built in Microsoft Visual C# and running on Microsoft .NET framework to process these ontological mapping rules. The engine processes rules as follows:

```
1: for all Step do
2: for all Action do
3: for all Script do
3: Execute SQWRL Query from the original IEC 61131-3 KB and Select
   variable Results from VAR
4: for all Results of VAR do
5: Replace Variable in the Template with SQWRL Query Results
6: Create an Instance in the target IEC 61499 KB with Template
   Content
7: end for
7: end for
8: end for
9: end for
```

Each step is responsible for processing an ontology node, and actions are handling the mapping between class, object properties, and data properties of this node. The engine will loop through steps and actions in each step. For each action, the engine will run the SQWRL query in the Script node which using SQWRL node as left-hand side and VAR node as right-hand side. Variables in the Template node are replaced with SQWRL query results. A new OWL instance is created in the IEC 61499 knowledge base which has a type and name defined in the Destination node with the content in the Template node. This new individual instance could be a new OWL class instance node in the target knowledge base, an object property instance linked to another node or a data property associated with some values. This new individual instance naming conversion is defined as

RootNodeType_SubNodeType_…_InstanceName

For example, if a program instance is created by the engine, it will be named as *Project_Task_Program_<ProgramName>*.

B. Migration Rules

In this subsection, the migration process will be described. In the approach by Sunder *et al.* [12], an IEC 61131-3 resource is mapped to an IEC 61499 resource and tasks and programs in the PLC are mapped to the IEC 61499 application. The cyclical execution behavior is also created in that approach, but no consideration of preemptive PLC runtime is presented.

In this particular example, a different approach is proposed with the complete consideration of preemptive execution semantics. The Rockwell ControlLogix PLC is selected as the source and the FBDK is used as the destination function block editor. Following the approach of [41], an IEC 61499 system configuration S is defined as a tuple

$$S = (Dev, Seg, App, Map) \tag{7}$$

where $Dev = \{Dev_1, Dev_2, \dots, Dev_n\}$ is a set of devices; *Seg* is the network segment for this IEC 61499 system configuration; $App = \{App_1, App_2, \dots, App_n\}$ is a set of applications defined. $Map = \{Map_1, Map_2, \dots, Map_n\}$ is a set of mappings of function blocks between applications and devices.

A system configuration is generated for each PLC project. The PLC name is mapped into the system configuration name. An action is created with mapping from *RSLogix5000Content* node in the PLC to the System node in the FB. The script has a content of *?b* that refers to a SQWRL variable in the template for selecting the project name.

The main operator of the SQWRL is sqwrl:select. It will fill a table that uses arguments as column names from the target knowledge base. The data property *TargetName* of the class *RSLogix5000Content* is selected as the name of the mapped system configuration. The *swrlb:stringConcat* in the query is one of the built-in functions from SWRL for construct string values. The results of the system name from the SQWRL query are stored in the variable ?b. The actual value replaces the *?b* in the data template of the content and this information is used by creating an ontology individual of the given class and its naming.

Mapping Rule 1: An IEC 61131-3 resource *Res* is mapped to an IEC 61499 device: $Res \rightarrow Dev$.

Next, a step is created for mapping from all IEC 61131-3 resources to IEC 61499 devices. The source of this action is the PLC resource instance (see Controller class in the PLC ontology) and the destination is the device instance of type RMT_DEV (Remove Device) in the FB ontology. The SQWRL query for selecting controller name as the new device name is

Algorithm 1 RSLogix5000Content(?a)

```
^Has_RSLogix5000Content_Controller(?a, ?b)
^Has_Controller_Name(?b, ?name)
-> sqwrl:select(?name)    (S.1)
```

As extra step, a main PLC scheduling function block is required for each mapped IEC 61499 device. A new device named *scheduling* is inserted into the system configuration as a migration action. The first action of this step is to recognize all periodic tasks and schedule those tasks properly into the main PLC scheduling function block. To construct the main PLC scheduling function block, the first SQWRL node of the Query in this Action is to find all periodic tasks and place them into the main scheduling function block

The above query retrieves names, types, and priorities of all tasks in a PLC configuration. *Has_Task_Name*, *Has_Task_Type*, *Has_Task_Rate and Has_Task_Priority*

Algorithm 2 Task(?Task) ˆ Has_Task_Name(?Task, ?Name)

```
ˆ Has_Task_Rate(?Task, ?Rate) ˆ Has_Task_Priority(?Task,
?Priority)
5 ˆ Has_Task_Type(?Task, ?Type)
ˆ swrlb:stringEqualIgnoreCase(?Type, "PERIODIC")
-> sqwrl:select(?Name, ?Priority, ?Rate)   (S.2)
```

are the data properties of the object-task. The *swrlb:stringEqualIgnoreCase* is the SWRL function for comparing string values. The *sqwrl:makeSet* is also a built-in collection operator to construct and manipulate sets. The main purpose of this operator is to support the closure operations for queries with negation or complex aggregation functionalities. The last operator *sqwrl:groupBy* is used to group sets of entities together. By using this query, all names, priorities, scan rate of tasks with type of periodic are listed in the resulting table.

After the data is ready, the query results need to be filled into the step template. A pair of event input and output as well as an enabling bit in the template of the main scheduling function block interface is created using the name of the task. Also an EC state named *<taskname>_DONE* is created for acknowledging the process complete signal from task schedulers and disabling that task. The REQ state is always inserted into the main scheduling function block. Emitting tasks triggers are created in the REQ state according to the orders of priorities. The REQ algorithm consists of three parts: time stamp calculation (?TimeStampPart), periodic task (?PTaskPart), and continuously task (?CTaskPart).

The data template of the content is given as

Algorithm 3 Timestamp:= Timestamp + 1;

```
IF Timestamp >= SCAN_TIME THEN
Timestamp:= Timestamp - SCAN_TIME;
?TimeStampPart
END_IF;
?PTaskPart
?CTaskPart
```

The script also consists of three SQWRL queries. The first SQWRL script is given as

Algorithm 4 Task(?Task)

```
ˆ Has_Task_Name(?Task, ?TaskName) ˆ
Has_Task_Type(?Task, ?Type)
ˆ swrlb:stringEqualIgnoreCase(?Type, "PERIODIC")
ˆ swrlb:stringConcat(?TimeStampPart, ?TaskName,
"_ENABLE:= FALSE;")   (S.3)
```

All names of periodic tasks are attached to "_ENABLE:= FALSE;" replace the ?TimeStampPart in the content of the data template as a single block. (For example: two periodic tasks P1 and P2 constructed a string "P1_ENABLE:= FALSE; P2_ENABLE:= FALSE;" replace the ?TimeStampPart)

The second script is to collect periodic tasks with their interval rate

Algorithm 5 Task(?Task)

```
ˆ Has_Task_Name(?Task, ?TaskName) ˆ
Has_Task_Type(?Task, ?Type)
ˆ Has_Task_Rate(?Task, ?Rate)
ˆ swrlb:stringEqualIgnoreCase(?Type, "PERIODIC")
ˆ swrlb:stringConcat(?NotTaskName, "& NOT(", ?TaskName,
"_ENABLE) & NOT(", ?TaskName, "_ENABLE_ONS)")
ˆ swrlb:stringConcat(?PTaskPart, "IF Timestamp >= ", ?Rate,
?NotTaskName,
"THEN\r\n," ?TaskName, "_ENABLE:= true;\r\n,"
?TaskName, "_ENABLE_ONS:= true;\r\n END_IF; \r\n").
(S.4)
```

The result algorithm in the EC State for a task named P1 with interval 20 ms will be

Algorithm 6 IF Timestamp >= 20 & NOT(P1_ENABLE)

```
& NOT(P1_ENABLE_ONS) THEN
P1_ENABLE:= true;
P1_ENABLE_ONS:= true;
END_IF
```

The third SQWRL script is similar to the second query. Instead of searching periodic tasks, continuously tasks are replaced.

The second action of this step is to find all continuous tasks from the code knowledge base by executing the following query:

Algorithm 7 Task(?Task) ˆ Has_Task_Name(?Task, ?Name)

```
ˆHas_Task_Priority(?Task, ?Priority)ˆHas_Task_Type(?Task,
?Type)
ˆ swrlb:stringEqualIgnoreCase(?Type, "CONTINUOUS")
-> sqwrl:select(?Name, ?Priority).   (S.5)
```

Similar to the periodic task, all names and priorities of continuous tasks are listed. When no periodic or higher priority continuous task is activated, the continuous task will keep executing.

After the system configuration is defined, the next level of IEC 61499 element is a device. An IEC 61499 device *Dev* is defined as a 2-tuple

$$Dev = (Rser, FBN) \tag{8}$$

where $Resr = \{Resr_1, Resr_2, \ldots, Resr_y\}$ is a set of resources used in the device; *FBN* is a function block network.

Mapping Rule 2: An IEC 61131-3 task *Task* is mapped to an IEC 61499 resource *Resr*: $Task \rightarrow Resr$.

According to the second rule, all IEC 61131-3 tasks are mapped to IEC 61499 resources. The source of this action is the PLC task instance (see Task class in the PLC ontology) and the destination is the resource instance of type EMB_RES (Embedded Resource) in the FB ontology. The PLC task name is used as the FB resource name.

After the resource is constructed, a task scheduling FB is necessary for each IEC 61499 resource to schedule programs in this task. Task scheduling function blocks are also added to the resource created previously and identical for both periodic and continuous tasks. In order to generate task schedulers, a list of programs and their execution orders is required for filling into the content template. A SQWRL query for selecting all programs in the execution order of a task is given as

Algorithm 8 Task(?Task)

```
^ Has_Task_Name(?Task, ?TaskName)
^ swrlb:stringEqualIgnoreCase(?TaskName, <TaskName>)
^ Has_ScheduledProgram (?Task, ?Program)
^ Has_ScheduledProgram_Name(?Program, ?Name)
-> sqwrl:select(?Name).   (S.6)
```

For each program, a separate EC state, EC state algorithm, and a pair of completed Boolean input and enable output are inserted into the task scheduling FB similar to the main scheduler FB. The EC state algorithm is generated in the same way as stated for the EC state of the main PLC scheduler FB.

The next level of the PLC code hierarchy is routines, functions, and function blocks inside a program. An IEC 61499 function block network *FBN* is defined as a 3-tuple

$$FBN = (FBI, EConn, DConn) \tag{9}$$

where $FBI = \{FBI_1, FBI_2, \ldots, FBI_n\}$ is a set of function block instances defined in a function block network; *EConn* is a set of event connections; *DConn* is a set of data connections.

An IEC 61499 composite function block *CFB* is defined as

$$CFB = (Interface', FBN) \tag{10}$$

where *Interface′* is IEC 61499 function block interface (including input/output events and data), *FBN* is a function block network;.

Mapping Rule 3a: An IEC 61131-3 program *Program* is mapped to an IEC 61499 composite function block *CFB* if there is some function or function block that is used inside a program: *Program → CFB (with function and function block).*

First, the program must be checked if any function or function block is invoked. The partial SQWRL queries are

Algorithm 9 Program(?Prog)

```
^ Has_Function(?Prog, ?Func)    (S.7)
And the second rule:
Program(?Prog)
^ Has_FunctionBlock(?Prog, ? Func).   (S.8)
```

Mapping Rule 3b: An IEC 61131-3 program *Program* is mapped to an IEC 61499 *BFB* if there is no function or function block that is used inside a program: *Program → BFB (without function and function block).*

If there is no function or function block calls in the original function block design, then a BFB is used to encapsulate the algorithm instead of a composite function block. The check for function or function block exists in the Rule 3a can be used in the opposite way to avoid duplicated generation of programs. Otherwise, the target code will be represented as multiple IEC 61499 function blocks. The rungs or lines before the instance call of a function or function block call, after the instance call of a function or function block call and between two functions or function blocks will be placed into another BFB. Those IEC 61499 function blocks are linked following the original execution order. A SQWRL query can be used to list all functions and function blocks in a PLC program. The generated function blocks are saved as OWL files back into the ontological knowledge base. Similar approach will be applied to the function or function block conversion if the IEC 61131-3 function or function block has nested structure inside.

The generation procedure for the program scheduling FB is almost equivalent to the task scheduler except the actual scheduling target are functions and function blocks inside the program instead of programs.

Now scheduling function blocks are all mapped. The execution order of the original PLC configuration is restored in the resulting function block system configuration. The architecture transformation of an IEC 61131-3 application to an IEC 61499 system is completed. However, there are several issues on the code level still need to be resolved in the generated system.

The first issue is how to map routines, functions, and function blocks into IEC 61499 manners. The migration rules reconstruct the program structure but still no code is filled into that structure yet. Several approaches of PLC systems redesign into function block systems are proposed by the authors in the previous work [39]. One of those approaches is to reuse PLC code in function block designs. In that approach, no PLC code modification is required when transformed into function block networks. Although other approaches are also feasible, reusing PLC code suits better for the migration process as it requires less human efforts and saving cost and work time.

An IEC 61499 BFB is defined as

$$BFB = (Interface', ECC, A\lg, IntData) \tag{11}$$

where *Interface′* is IEC 61499 function block interface; *ECC* is representing an execution control chart in the BFB; *Alg* is a set of algorithms associated with EC states; *IntData* is a set of internal variables only used in this BFB.

PLC program can be written in one of the four programming languages of the IEC 61131-3 standard or a graphical language SFC. For a program written in ladder logic diagram (LD), instruction list (IL) or structure text (ST), the code can be placed directly into an algorithm of a BFB. However, there is one exception—the case, when another IEC 61131-3 function or function block is invoked in this program. This requires calling an instance of a function block inside a BFB. This is not supported by the existing IEC 61499 tools and not specified in the standard itself. The idea of the solution is illustrated in the Fig. 14.

If there is no function or function block instance called in the program, the entire logic can be placed into a Basic FB. Otherwise those function or function block instances are mapped to an inherited CFB and logics around are mapped into separated BFBs.

An IEC 61499 execution control chart *ECC* is defined as

$$ECC = (ECstate, ECTrans, ECTCond, L) \tag{12}$$

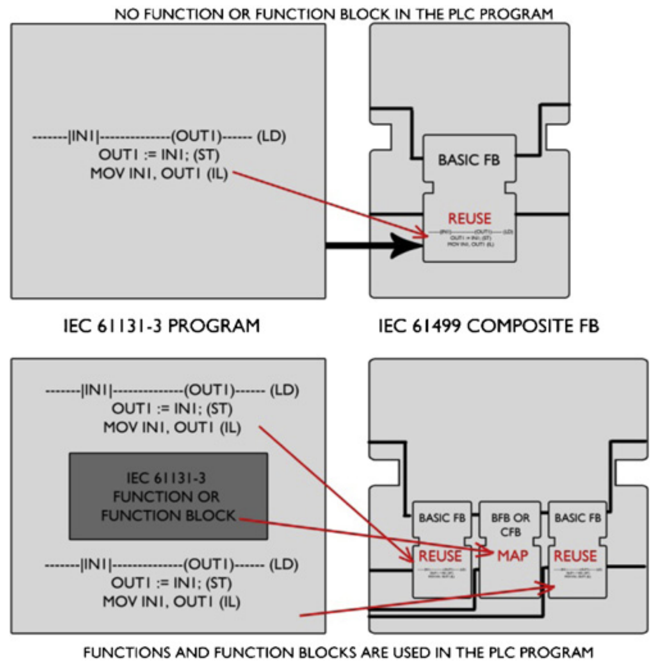

NO FUNCTION OR FUNCTION BLOCK IN THE PLC PROGRAM
-------|IN1|--------------(OUT1)------ (LD)
OUT1 := IN1; (ST)
MOV IN1, OUT1 (IL)
BASIC FB
REUSE
IEC 61131-3 PROGRAM
IEC 61499 COMPOSITE FB
IEC 61131-3 FUNCTION OR FUNCTION BLOCK
BFB OR CFB
MAP
FUNCTIONS AND FUNCTION BLOCKS ARE USED IN THE PLC PROGRAM

Fig. 14. Reuse PLC code in IEC 61499 function blocks for LD, IL, and ST.

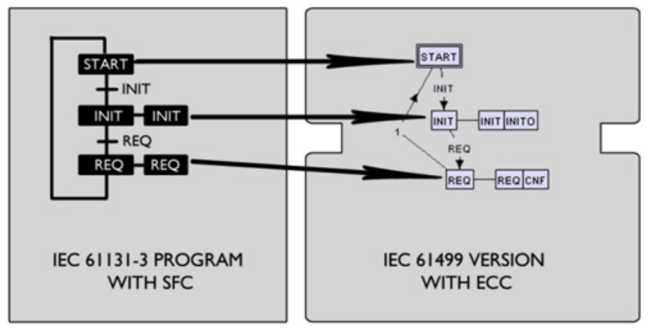


Fig. 15. IEC 61131-3 SFC mapping to IEC 61499 ECC.

where *ECstate* is a set of EC states; *ECTrans* is a set of EC transitions; *ECTCond* is a set of EC transition conditions; L: *ECTrans* → *ECTCond* is a function assigning EC transition conditions to EC transitions. **Mapping Rule 4:** An IEC 61131-3 SFC program *Program* is mapped to an ECC inside the IEC 61499 BFB: *Program* → *ECC (if program is written in SFC).*

If the original program is written in the SFC, the SFC can be converted to the ECC inside a BFB [42]. Each SFC step is mapped to an EC state. SFC transitions, transition conditions, step algorithms are mapped to EC transition, EC transition conditions and EC state algorithm correspondingly by using SQWRL queries and the transformation engine as showing in Fig. 15. However, parallel SFC steps are allowed in IEC 61131-3 but no concurrent ECC is defined in the IEC 61499 standard. This is not considered in this paper; however a solution is proposed by Riedl *et al.* [42].

Mapping Rule 5: An IEC 61131-3 FBD program *Program* is mapped to an IEC 61499 composite function block *CFB*: *Program* → *CFB (if program is written in FBD).*

If the original program is written in the function block diagram language, it can be converted to the function block

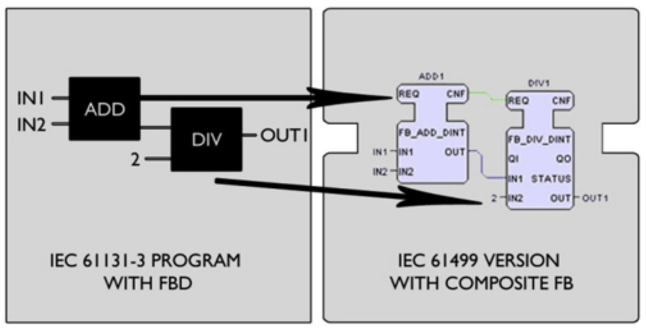


Fig. 16. IEC 61131-3 FBD mapping to IEC 61499 Composite FB.

network of IEC 61499. Each function block in the FBD of IEC 61131-3 is converted to an IEC 61499 function block and connections between function blocks are easily established. All function blocks converted from the original PLC version must be in a single event chain in order to be executed once in each scan in sequential order as illustrated in the Fig. 16. This can be achieved by applying SQWRL queries selecting all original function blocks in their original order and inserting the corresponding function blocks to the IEC 61499 application.

The other issue of the migration process is related to treating the global and program variables used in the PLC code. There are two levels of variables in the PLC program: controller (global) variables and program (local) variables. A controller variable can be accessed from anywhere in the PLC configuration. A program variable can only be accessed from a particular program. There is no global variable concept in the IEC 61499 standard as it is designed for distributed systems control. The only place where a variable might be stored is a BFB. The alternative choice is to build a SIFB to access some external data sources. But the SIFB is implementation dependent that means a different SIFB must be created manually for every IEC 61499 platform. For the automatic migration process, using BFBs to store variables is more suitable. In order to access global variables from any level in the IEC 61499 hierarchy, a pair of Publish and Subscribe function block is inserted. As indicated in Fig. 17, a BFB is used as the global variable storage. Each variable has a data input publishing its value and subscribing writing requests over the entire PLC configuration. The same procedure is repeated again for all global variables that are listed by using SQWRL queries of selecting all global variables. Publish and subscribe function blocks are named as variable names they refer to.

The data storage function block is also suitable for program variables. The same procedure is applied with SQWRL queries selecting all program variables from the knowledge base.

At this stage, mapping the original PLC code to the function block version is defined completely in the transformation procedure. By using the migration engine, all artifacts of the original PLC code are transformed into the function block knowledge base. Finally, code generation applied to the knowledge base will deliver IEC 61499 code.

VI. Case Study on Baggage Handling System

An inbound BHS as shown in Fig. 18 is used as the case study for migrating from IEC 61131-3 PLCs to IEC 61499

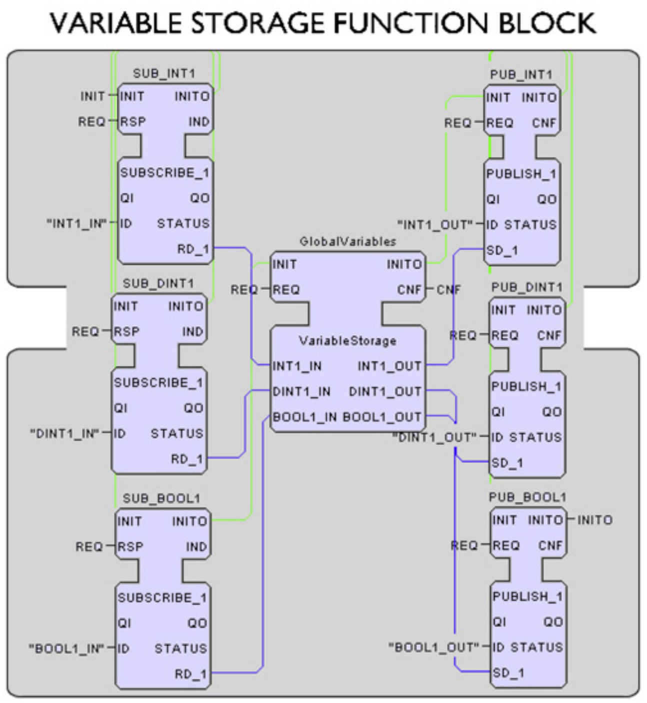


Fig. 17. Global/local variable composite function block.

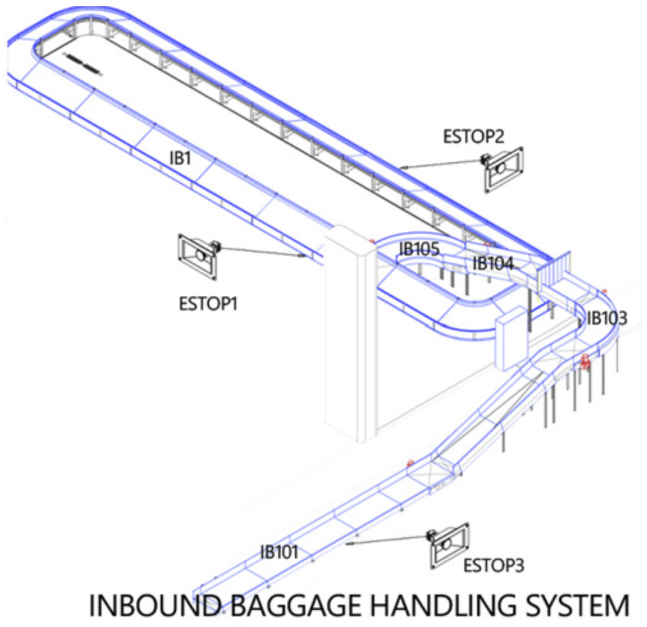


Fig. 18. Case study inbound BHS layout.

function blocks. The original PLC program is written in the Rockwell ControlLogix PLC. There are five conveyors and one inbound baggage carousel in the system. Bags are inducted from the IB101 take-away conveyor and merged into the IB1 carousel. There are also three emergency stops located around the system.

The original PLC code structure is shown in Fig. 19. There are two tasks scheduled in the PLC. The *FastTask* is a periodic task which executes every 25 ms. A program *FastProgram* is associated with this task. There is one merge control and three emergency stop controls scheduled in the program. Also

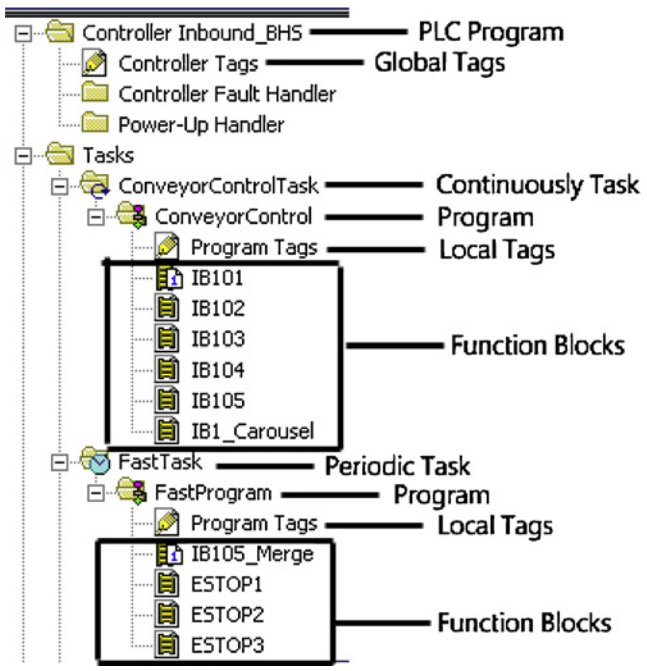


Fig. 19. Original PLC configuration structure.

there is a continuously task *ConveyorControlTask* which has a program *ConveyorControl* in the PLC configuration. Inside the program, there are six conveyor control function blocks schedule. Each of them is controlling a physical conveyor in the system. Inside all function blocks, there is no nested level of functions or function blocks invoked.

The resulting IEC 61499 system configuration achieved by applying the migration process is given in Fig. 20.

The resulting system configuration has three resources. The periodic task *FastTask* is mapped to the resource *FastTask*. Each function block in the PLC program *FastProgram* is converted to an IEC 61499 function block. Those function blocks are linked according to the order of their counterparts in the PLC program. The resource *ConveyorControlTask* includes all six conveyor control function blocks in the order of the original PLC program. A new resource *Scheduling* has a main PLC scheduling FB, two task scheduling FBs, and two program scheduling FBs. The enable event is raised by the program scheduling FBs to either *FastTask* resource or *ConveyorControlTask* resource. Once all function blocks in the *FastTask* resource are executed, an acknowledgement event is send back to the program scheduling FB to indicate the execution of the periodic task is accomplished. The main scheduling function block will switch to the continuously executed task until the predefined execution cycle for the periodic task arrives again.

This case study is supported by the automatic migration tool developed by the authors. The tool imports PLC code in the XML format (for example, PLCOpen XML [37] and Rockwell [38]) and converts it into the IEC 61131-3 knowledge base. Then those instances in the IEC 61131-3 knowledge base are mapped to the IEC 61499 knowledge base by the migration engine. Finally the FB version of the code is generated. The complexity of this migration process is linear to the size of the original PLC program. An 8MB PLC XML file takes approximately 5 min to complete the entire migration process.

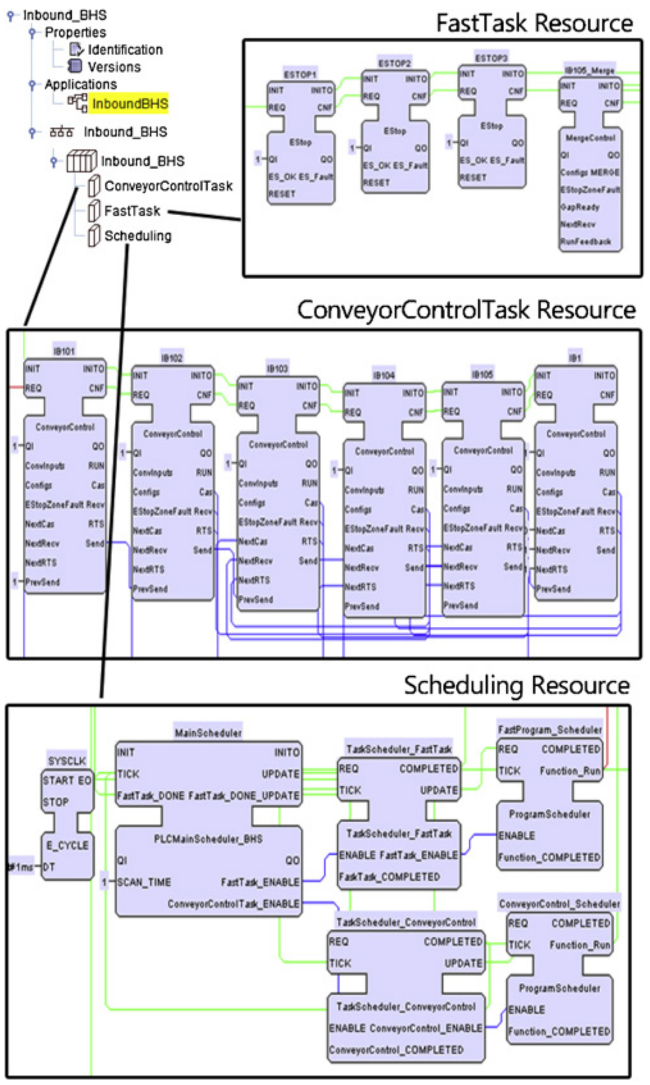


Fig. 20. Generated function block configuration structure and main PLC scheduler FB algorithm.

VII. Conclusion

A new ontological-based migration procedure was proposed in this paper that can be applied to IEC 61131-3 compliant PLC source code to transform it to IEC 61499 platforms. In order to achieve that, a formal model of IEC 61131-3 application particularly for migration and its execution model was defined. Also the mapping rules between migration models are provided. Code import, automatic mapping, and transformation between standards and code generation were achieved by using knowledge-base queries. The main advantage of this approach is reliance on the semantic web technologies. The developed migration tool can use the standard S(Q)WRL engine that is configurable by the rules. This approach is more flexible and less resource consuming in development as compared to hard coding the transformation rules. The ontological knowledge base provides a higher abstract level view of the migration process. A case study was conducted to prove the automatic migration from IEC 61131-3 PLCs to IEC 61499 function blocks is feasible.

Future work will also include detailed comparison of the obtained IEC 61499 code with the code designed manually using the approaches proposed in [21] both in terms of performance and code maintainability.

The known limitations of this approach are as follows. First, the target IEC 61499 platforms must support all instructions used in the original IEC 61131-3 PLC programs in order to reuse the entire PLC program. Second, the source IEC 61131-3 platforms must support export to the XML file format. IEC 61131-3 function blocks with source protection enabled must be available as SFIB in the target IEC 61499 runtime. The interface of those IEC 61131-3 function blocks must be as available in XML format. Also the generated scheduling system can only be interrupted after the current function block is completely executed. The IEC 61131-3 PLC will interrupt at instructions level and resume from there. Finally, the minimum tick in IEC 61499 is one millisecond due to the platform used. The PLC normally executes at microsecond's level. This is caused by the FBRT IEC 61499 runtime running on a PC-based controller with nonreal time operation systems (The minimum time scale on MS Windows is 1 ms). The E_CYCLE SIFB can generate events at microsecond's level if the target controller is running a real time operation system instead.

Future work will also concern with semantic analysis of generated function block systems and correction of errors introduced during the migration process. The accuracy of the generated IEC 61499 systems will be improved by introducing ECC refactoring [21]. Finally, more PLC platforms and complicated examples will be tested with the proposed migration rules.

References

[1] *IEC 61499, Function Blocks*, Int. Std., 1st ed., 2005

[2] V. Vyatkin, "IEC 61499 as enabler of distributed and intelligent automation: State of the art review," *IEEE Trans. Ind. Informat.*, vol. 7, no. 4, pp. 768–781, Nov. 2011.

[3] *IEC 61131-3, Programmable Controllers—Part 3: Programming Languages*, Int. Std., 2nd ed., 2003.

[4] A. Zoitl, T. Strasser, C. Sunder, and T., Baier, "Is 61499 in harmony with IEC 61131-3?," *IEEE Ind. Electron. Mag.*, vol. 3, no. 4, pp. 49–55, Dec. 2009.

[5] T. Strasser, A. Zoitl, J. Christensen, and C. Sunder, "Design and execution issues in IEC 61499 distributed automation and control systems," *IEEE Trans. Syst., Man, Cybern. C, Appl. Rev.*, vol. 41, no. 1, pp. 41–51, Jan. 2011.

[6] T. Strasser and R. Froschauer, "Autonomous application recovery in distributed intelligent automation and control systems," *IEEE Trans. Syst., Man, Cybern. C, Appl. Rev.*, vol. 42, no. 6, pp. 1054–1070, Nov. 2012.

[7] N. Cai, M. Gholami, L. Yang, and R. Brennan, "Application-oriented intelligent middleware for distributed sensing and control," *IEEE Trans. Syst., Man, Cybern. C, Appl. Rev.*, vol. 42, no. 6, pp. 947–956, Nov. 2012.

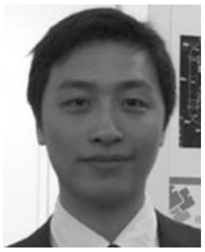
[8] J. Peltola, J. Christensen, S. Sierla, and K. Koskinen, "A migration path to IEC 61499 for the batch process industry," in *Proc. 5th Int. Conf. Ind. Inform.*, vol. 2. 2007, pp. 811–816.

[9] T. Hussain and G. Frey "Migration of a PLC controller to an IEC 61499 compliant distributed control system: Hands-on experiences," in *Proc. IEEE Int. Conf. Robot. Autom.*, Apr. 2005, pp. 3984–3989.

[10] M. Wenger, A. Zoitl, C. Sunder, and H. Steininger, "Transformation of IEC 61131–3 to IEC 61499 based on a model driven development approach," in *Proc. 7th IEEE Int. Conf. Ind. Inform.*, 2009, pp. 715–720.

[11] M. Wenger, A. Zoitl, C. Sunder, and H. Steininger, "Semantic correct transformation of IEC 61131-3 models into the IEC 61499 standard," in *Proc. IEEE Conf. Emerging Technol. Factory Autom.*, Sep. 2009, pp. 1–7.

[12] C. Sunder, M. Wenger, C. Hanni, I. Gosetti, H. Steininger, and J. Fritsche, "Transformation of existing IEC 61131-3 automation projects into control logic according to IEC 61499," in *Proc. IEEE Int. Conf. Emerging Technol. Factory Autom.*, Sep. 2008, pp. 369–376.
[13] N. Bauer, R. Huuck, B. Lukoschus, and S. Engel, "A unifying semantics for sequential function charts," *LNCS*, vol. 3147, no. 2004, pp. 400–418, 2004.
[14] F. Basile, P. Chiacchio, and D. Gerbasio, "Progress in PLC programming for distributed automation systems control," presented at 9th Int. Conf. Ind. Inform., 2011, pp. 621–627.
[15] K. M. Goh and W. Dint, "Iterative knowledge based code generator for IEC 61499 function block," presented at the IEEE Region 10 Conf. (TENCON), Singapore, 2009.
[16] M. B. Younis and G. Frey, "A formal method based re-implementation concept for PLC programs and its application," in *Proc. IEEE ETFA*, Sep. 2006, pp. 1340–1347.
[17] G. Weisenborn, "Method for converting a programmable logic controller hardware configuration and corresponding control program for use on a first programmable logic controller to use on a second programmable logic controller." U.S. Patent No. 5142469, 1992.
[18] A. Fay, "A knowledge-based system to translate control system applications," *Eng. Appl. Artificial Intel.*, vol. 16, nos. 5–6, pp. 567–577, 2003.
[19] A. Mader and H. Wupper, "What is the method in applying formal methods to PLC applications?," in *Proc. ADPM*, 2000, pp. 165–171.
[20] J. Chouinard. (2012). "Distributed systems development with IEC 61499 in ISaGRAF," [Online]. Available: http://www.ece.auckland.ac.nz/~vyatkin/iec61499day/index.html
[21] V. Dubinin and V. Vyatkin, "Semantics-robust Design Patterns for IEC 61499," *IEEE Trans. Ind. Informat.*, vol. 8, no. 2, pp. 279–290, May 2012.
[22] G. Cengic and K. Akesson, "On formal analysis of IEC 61499 applications, part B: Execution semantics," *IEEE Trans. Ind. Informat.*, vol. 6, no. 2, pp. 145–154, May 2010.
[23] W. Dai, V. Vyatkin, and V. Dubinin, "Semantic analysis of IEC 61499 systems using automatically generated ontological models," *IEEE Trans. Ind. Informat.*, 2012, DOI 10.1109/TII.2012.2235450, still in print.
[24] H. Y. Li, P. Roop, V. Vyatkin, and Z. Salcic, "A synchronous approach for IEC 61499 function block implementation,"*IEEE Trans. Comput.*, vol. 58, no. 12, pp. 1599–1614, Dec. 2009.
[25] (2011). Ontology general definition [Online]. Available: http://semanticweb.org/wiki/Ontology
[26] F. Badder, D. Calavanese, D. L. McGuinness, D. Nardi, and P. F. Patel-Schneider, *The Description Logic Handbook, Theory, Implementation and Applications,* 2nd ed. Cambridge, U.K.: Cambridge University Press, 2007.
[27] M. O'Connor and A. Das, "SQWRL: A query language for OWL," in *Proc. 5th Int. Workshop OWLED,* vol. 529, 2009.
[28] E. Prud'hommeaux and A. Seaborne. (2008). *SPARQL Query Language for RDF* [Online]. Available: http://www.w3.org/TR/rdf-sparqlquery/
[29] *FBDK—Function Block Development Kit* [Online]. Available: http://www.holobloc.com/
[30] nxtStudio by nxtControl GmbH. (2009, Jun.). "nxtControl—Next generation software for next generation customers" [Online]. Available: http://www.nxtcontrol.com
[31] (2011). 4DIAC, An open source IEC 61499 IDE and runtime [Online]. Available: http://www.fordiac.org
[32] (2011). FBench, An open source IDE for IEC 61499 [Online]. Available: http://www.ece.auckland.ac.nz/~vyatkin/fbench/
[33] CORFU. (2010). Function block development environment [Online]. Available: http://seg.ee.upatras.gr/corfu/dev/index.htm
[34] A. Zoitl and H. Prahofer, "Building hierarchical automation solutions in the IEC 61499 modeling language," presented at 9th IEEE Int. Conf. Ind. Inform., 2011, pp. 557–564.
[35] *ISaGRAF Workbench* [Online]. Available: http://www.isagraf.com
[36] V. Vyatkin and J. Chouinard, "On comparisons the ISaGRAF implementation of IEC 61499 with FBDK and other implementations," in *Proc. 6th IEEE INDIN*, Jul. 2008, pp. 289–294.
[37] *PLCOpen—the worldwide community related to IEC 61131-3* [Online]. Available: http://www.plcopen.org
[38] *Rockwell Automation—Major PLC vendor* [Online]. Available: http://www.rockwellautomation.com
[39] W. Dai and V. Vyatkin, "Redesign distributed PLC control systems using IEC 61499 function blocks," *IEEE Trans. Autom. Sci. Eng.*, vol. 9, no. 2, pp. 390–401, Apr. 2012.
[40] SWRL: A Semantic Web Rule Language Combining OWL and RuleML [Online]. Available: http://www.w3g.org/Submission/SWRL/
[41] V. Vyatkin and V. Dubinin, "Refactoring of execution control charts in basic function blocks of the IEC 61499 standard," *IEEE Trans. Ind. Informat.*, vol. 6, no. 2, pp. 155–165, May 2010,.
[42] M. Riedl, C. Diedrich, and F. Naumann, "SFC in IEC 61499," in *Proc. IEEE Int. Conf. Emerging Technol. Factory Autom.*, Sep. 2006, pp. 662–666.
[43] (2012). Semantic Web [Online]. Available: http://www.w3.org/standards/semanticweb/



Wenbin (William) Dai (S'09–M'13) received the Bachelor of Engineering (Hons.) degree in computer systems engineering from the University of Auckland, Auckland, New Zealand in 2006. He received the Ph.D. degree in electrical and electronic engineering from the Department of Electrical and Computer Engineering, The University of Auckland, Auckland, New Zealand, in 2012.

He is currently a Post-Doctoral fellow with Luleå University of Technology, Luleå, Sweden. He has been also a Software Engineer with Glidepath Limited—a New Zealand-based airport baggage handling system provider, since 2007. He has been involved in a number of airport baggage handling system and parcel sortation system projects in New Zealand, Australia, Canada, China, Africa, the Middle East, and South America. His responsibilities include design and development of PLC control and SCADA/HMI for those systems. His current research interests include IEC 61131-3 PLC, IEC 61499 function blocks, distributed control systems, industrial fieldbus communication protocol, SOA, and Internet of Things in industrial automation.

Victor N. Dubinin received the Diploma degree in computer science and the Ph.D. degree from the University of Penza, Penza, Russia, in 1981 and 1989, respectively.

From 1981 to 1989, he was a Researcher and from 1989 to 1995, he was a Senior Lecturer at the University of Penza. Since 1995, he has been an Associate Professor with the Department of Computer Science at the University of Penza. In 2011, he held a Visiting Researcher position at the University of Auckland, New Zealand. His current research interests include formal methods for specification, verification, synthesis, and implementation of distributed and discrete event systems.

Dr. Dubinin was awarded DAAD-grants to work as a Guest Scientist at Martin-Luther-University, Halle-Wittenberg, Germany in 2003, 2006, and 2010.

Valeriy Vyatkin (M'03–SM'04) received the Engineering degree in applied mathematics in 1988 from the Taganrog State University of Radio Engineering (TSURE), Taganrog, Russia. Later he received the Ph.D. and Dr. Sci. degrees in 1992 and 1998, respectively, from the same university, and the Dr. Eng. degree from the Nagoya Institute of Technology, Nagoya, Japan, in 1999.

He is a Chaired Professor (Ämnesprofessor) of Dependable Computation and Communication Systems at the Luleå University of Technology, Luleå, Sweden, and Visiting Scholar at Cambridge University, U.K., on leave from The University of Auckland, Auckland, New Zealand, where he has been Associate Professor and Director of the InfoMechatronics and Industrial Automation lab (MITRA) at the Department of Electrical and Computer Engineering. His previous faculty positions were with Martin Luther University of Halle-Wittenberg in Germany (Senior Researcher and Lecturer from 1999 to 2004), and with TSURE (Associate Professor, Professor from 1991 to 2002). His current research interests include in the area of dependable distributed automation and industrial informatics, including software engineering for industrial automation systems, distributed architectures and multiagent systems applied in various industry sectors, including Smart Grid, material handling, building management systems, and reconfigurable manufacturing. He is also active in research on dependability provisions for industrial automation systems, such as methods of formal verification and validation, and theoretical algorithms for improving their performance.

Dr. Vyatkin was awarded the Andrew P. Sage Award for the best IEEE TRANSACTIONS paper in 2012.