Please cite as follows: W. Dai, V. Dubinin, V. Vyatkin, "Automatically Generated Layered Ontological Model for Semantic Analysis of Component-Based Control Systems", IEEE Transactions on Industrial Informatics, 2012, doi: 10.1109/TII.2012.2235450

Automatically Generated Layered Ontological Models for Semantic Analysis of Component-Based Control Systems

Wenbin (William) Dai, *IEEE Member*, wdai005@aucklanduni.ac.nz, Victor Dubinin, victor_n_dubinin@yahoo.com, Valeriy Vyatkin, *Senior IEEE Member*, vyatkin@ieee.org

***Abstract* — The IEC 61499 standard is designed for distributed control and proposes new visual form of programming using block diagrams with embedded state machines and unlimited hierarchical nesting and distribution across networking devices. Such visual programs require new methods of automatic syntactic and semantic analysis. This paper proposes a new approach to semantic analysis using multiple-layered ontological knowledge representation and rule-based inference engine. Its working is demonstrated on example.**

***Index Terms* — Component-based software architecture, IEC 61499 Function Blocks, Embedded Control Systems, Ontology, Knowledge Base, OWL, SWRL, SQWRL, Description Logic (DL), Semantic Analysis, Ontology Reasoning.**

I. Introduction

The component-based function block architecture of IEC 61499 standard [1], [2] targets the system-level design of complex distributed automation systems. It proposes a new visual form of programming using block diagrams with embedded state machines and unlimited hierarchical nesting, being distributed across networking devices. The basic design artefact of this architecture is an event-triggered function block. The event interface and distribution are the key differences of this architecture from the function blocks used in automation systems (as standardized in the IEC 61131-3 standard [3] for programmable logic controllers (PLC)). The component-based architecture of IEC 61499 provides a self-explanatory language with a better overview of the entire system for automation software design.

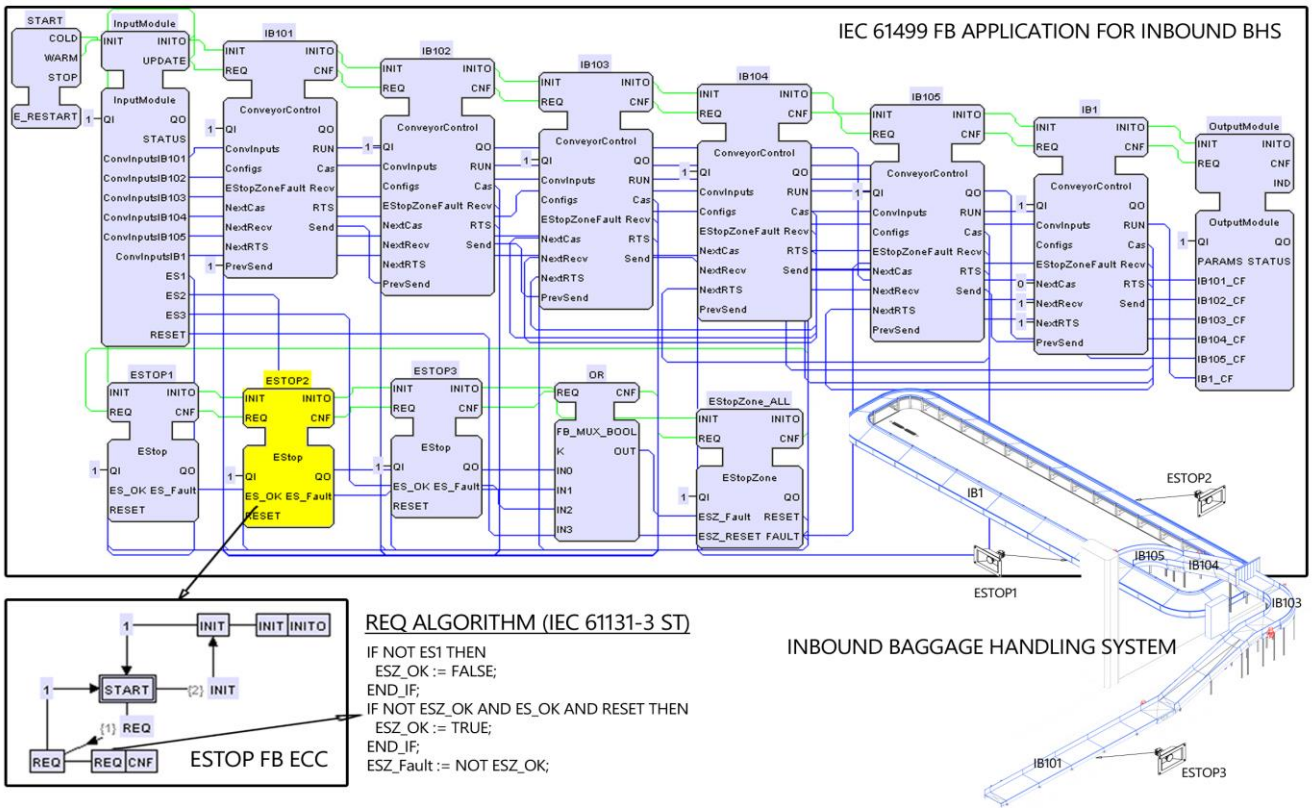


Fig. 1. An example of a block-diagram programming using the IEC 61499 function blocks.

Fig. 1 illustrates an example of a simple airport inbound baggage handling system, fully programmed with function blocks. The example system consists of six conveyors including one baggage claim carousel and three emergency stop control stations. Each physical device is controlled by a software component - function block. The top part of the function block network in

Fig. 1 represents the conveyor chain and the bottom part is used for emergency stop zone control. However, along with numerous benefits [4], this visual design approach creates new challenges for syntactic and semantic analysis of such programs.

In traditional PLC programming languages, the syntactic (and partially semantic) check is done by software tools as a part of compilation or editing process. The IEC 61499 function block standard uses XML representation format for all design artefacts. The use of XML technologies simplifies the implementation of syntax checkers which can be delegated to the standard XML parsers. Such tools are easily extendable that is achieved by changing the corresponding XML schema in case of syntax extensions. However, the XML technology is of little help when it comes to spotting semantic problems, so an application with semantic issues could pass the syntactic check.

As one can appreciate even from the simple example presented in Fig. 1, the function block designs can be quite complex thus hard to debug and validate. The semantic analysis support can increase performance of developers and dependability of automation systems, and eventually help with adoption of the component-based software architecture technology in the automation industry. A small example of semantic check in the function block context is implemented in the experimental FBDK tool [5] whose editor prohibits creating loops in state machines which do not include a transition triggered by an event input. Although correct syntactically, such loops can cause deadlocks in the program execution. However, the implementation of this semantic rule is hard coded in the tool and cannot be extended or modified if the syntax or semantics of the state machines changes, as it was the case recently with the second edition of the standard introduced. Unfortunately, no semantic check functionality is implemented in any commercial component-based PLC software tools such as ISaGRAF Workbench [6] or NxtStudio [7].

In this paper, a novel methodology of *extendable* semantic check for component-based software architecture - function blocks is developed. Its configurability is akin to the XML-supported syntax check, but implementation is based on the semantic Web technologies such as ontologies, used to represent the semantic model of programs and extensible set of semantic rules. The proposed semantic analysis methodology can be conveniently integrated into the existing component-based PLC development environments.

The paper is structured as follows: Section II provides an introduction into the related Semantic Web technologies. The related research works are reviewed in section III. The contribution of this work is summarized in the beginning of section IV. In sections V, a layered ontology design for software components is proposed with detailed generation of an ontology model from XML schema and document type definition (DTD) as well as automatic import of function block systems into the searchable knowledge base. Semantic corrections and extensions to the ontology based knowledge base are listed in Section VI along with the rules for semantic analysis. In Section VII, a case study of applying semantic analysis to a baggage handling system example is presented. The tool developed for component-based software architecture semantic analysis is described in section VIII. The paper is concluded with a summary and future works plan.

II. Syntactic and Semantic Analysis For Function blocks using Ontology Model

The first part of the analysis chain is the syntax check. As the function block code is represented in XML, the corresponding document type definition (DTD) is provided as well. The DTD defines all possible elements and sub-elements which are allowed to be used in the function block XML documents. Another form of XML syntax description is XML Schema (XSD). Unlike DTD, XSD uses XML to describe syntax of custom XML documents. In addition to DTD, XSD supports all data types including the user defined ones, whereas DTD only supports generic character types CDATA and PCDATA [8]. In this work, both XSD and DTD can be used as syntax description sources. The DTD version is used as an example.

However, the behavioural (semantic) properties of component-based software architecture artefacts cannot be verified by syntactic checking. Some implementations of general purpose programming languages use semantic rules hard-coded in compilers, but this approach is not flexible. Any change of the input language, or introduction of new semantic rules, will require change of the compiler.

In this work we propose a more flexible approach where semantic checker is using an external (and thus extendable) set of semantic rules. The rules refer to the knowledge model representing the related concepts of the function blocks architecture. More complex rules can be built on top of existing ones.

The ontology mechanism [9] is used as a proper representation of such a knowledge base. The “Ontology” concept, originated in philosophical studies of the nature of being, existence or reality, is now widely applied in the Semantic Web programming and service-oriented architectures (SOA). Recently this concept was introduced into the automation and control area [11-12] where it has been majorly applied to describe processes in manufacturing and processing plants.

In this work we propose a novel way of using ontologies to describe properties of automation software systems designed according to the component-based software architecture. Compared to the hard-coded semantic analysis, the approach based on the generic ontology model for the component-based software architecture can handle automatically any change of the input language without modifying the analyser’s core code. The semantic rules using the ontology are both human and machine readable and self-explanatory.

Advantages of using ontology in the semantic analysis for function block systems are as follows:

Firstly, ontology is a formal mechanism for describing semantic properties which have been originally represented in words. The formal reasoning languages and tools can be applied to check those properties. This increases the reliability of the analysis results. Also it provides a high level view of all properties and relationships inside function block systems.

Secondly, the ontologies are extendable that allows describing partial knowledge which can be extended in the future. By using languages based on the description logic (DL) [13] (for example, OWL DL) and Semantic Web Rule Language (SWRL) [14], the additional ontology definition can be merged into an existing ontology model seamlessly. SWRL extends the set of OWL axioms to include Horn-like rules. It thus enables Horn-like rules to be combined with an OWL knowledge base.

Finally, expressions defined in DL and SWRL for ontology based semantic analysis can be changed easily without changing the semantic analysis engine. This engine can be utilized for other XML based languages including IEC 61499 XML.

The need for semantic check flexibility is motivated by several reasons, the main of which are as follows:

- New versions of the standard-compliant software may redefine the semantic correctness, and the corresponding checkers would need to be upgraded;
- The rules may define correctness in the context of particular design patterns. New design patterns are constantly being introduced which requires to add new semantic correctness definitions;
- Last but not least, the standard itself is a living organism which undergoes maintenance, leading to certain syntax and semantics changes.

III. Related Works

This paper extends the work by Dubinin et al. [10] that presented a technique for semantic analysis of IEC 61499 based on ontological model that can be regarded as a direct predecessor of this paper. While the concept has been soundly demonstrated in [10], it was based on a manually created ontological model. It is hard to create software tools implementing it on systematic basis since instance data have to be entered manually into the ontological knowledge base.

The new approach of the present paper uses automatically generated ontological models with data instances which paves the way to implementation of software tools. Also this paper presents new execution semantic rules of execution control chart (ECC) in basic function blocks and other complicated rules, not covered in [10].

Along with [10], a similar idea of applying ontological mechanisms for semantic analysis appeared in the work by Arakawa [15], but in a completely different domain: it proposed a method for analyzing natural language texts using ontologies.

Further in this Section a number of related works is discussed, especially those related to the main enabling technologies of this research.

Ontologies are widely used in the software engineering domain [11], [12], [16], [19] and [20]. However there is no previous work about the using of ontologies for the semantic analysis of programming and domain specific languages.

The feasibility of applying semantic web and service-oriented architecture into automation industry has been discussed by Jammes and Smit in [17]. The work is motivated by the challenges of interoperability, scalability, plug-and-play connectivity and seamless integration. The service-oriented architecture using the Web services standards is applied to automation systems. That paper proves that the use of SOA and Web services standards can enhance the intelligence of automation systems.

Goh and Dint in [18] 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 extendable 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 pre-defined rules are satisfied, the related block of code will be generated and data types and connections will be also inserted. However, this XML and EBNF based approach is not convenient for configuring the IEC 61499 systems manually. Besides, the code template must be pre-defined in the knowledge base manually and cannot be easily modified when the code template is changed.

Orozco and Lastra [28] illustrate the idea of how semantics can be added to the Function Blocks reference models of the standard IEC 61499 by using ontologies. But the intention of this paper was to use semantic descriptions of FBs for automatic searching and discovery of FBs in applications based on the web services. Also the FB ontology model presented in this paper is not detailed enough for semantic analysis.

In [10], an ontology for semantic analysis of IEC 61499 compliant systems is proposed. The function block type ontology model includes basic, composite and service interface function blocks and system configurations. The ontology model for any function block type includes a model of its interface. Along with that, the ontology model of a basic function block includes the Execution Control Chart (ECC) ontology model. In the ECC ontology model, EC state, EC algorithm, EC Transition and EC Transition conditions are defined. The composite function block ontology model includes references to the component function block instances along with models of event and data connections. Finally, the system configuration ontology model contains devices, resources, applications, connections, mappings and network segments and their object and data properties. That paper provides simple examples of semantic analysis for IEC 61499 files using description logic and SWRL. A semantic analysis tool using a Protégé plug-in is developed for automatic semantic checking.

IV. Layered IEC 61499 Ontology Design

The IEC 61499 ontology in [10] was developed manually leaving the questions of how adequate is it to the text of the standard and, especially, to particular implementations, which may slightly deviate from the standard or may extend its insufficiently defined parts. This work attempts to overcome this shortage by proposing a layered approach to structuring the ontology. The base level of the ontology is automatically generated from the XML schema that captures most of syntactic properties and is used directly by function block tools for syntactic analysis. This approach promises to have less discrepancies between the code syntax supported by a tool and its semantic analyser. Moreover, the layered approach

promises better extensibility of the ontology, or possibility to customize it for a particular dialect or design pattern.

According to the IEC 61499 standard, XML is used to define three classes of artefacts: (1) Library Elements which include elements from system configuration, devices, resources, applications and sub-applications, function block types, adapter types, network segments and mapping applications to resources; (2) Function Block Management Commands which define the protocol used for communication between the IEC 61499-compliant devices, and (3) Data Types allowed in IEC 61499 artefacts.

The proposed ontology for this component-based software architecture includes three corresponding nodes at the top level of its hierarchy. From those nodes, all items, defined in the original XML DTDs, will be automatically transformed in a hierarchical structure. At this stage, the generated ontology model is totally based on syntactic rules with very limited semantic information that includes only the quantity of items that can exist in the system. Also the rule is mostly based on library elements. All examples presented in this paper will be belonging to library elements.

After developing the ontology model capable of accurately representing the syntax, the next step is to add there some semantic rules. We propose three types of rules to verify an IEC 61499 system.

(i) The first semantic layer consists of basic and simple rules to check that all the types (both function block and data types) are matched correctly. For example, in implementations where typecasting is not supported, all data elements connected via arcs shall be identical. More generally, the implementation dependent typecasts can be represented as semantic rules. This ensures all events and data variables are correctly defined and properly used.

(ii) The next layer of rules aims at achieving the correct execution semantics of the FB system or of a single FB. There are separate sets of rules for each execution semantics of IEC 61499 [21].

(iii) The final category of the semantic rules is to check the compliance with particular design patterns or absence of known semantic problems.

The rules are defined in terms of the Semantic Web Rule Language (SWRL) [14] and can be verified by the standard ontology reasoner or using the Semantic Query-Enhanced Web Rule Language (SQWRL) [22] query engine.

V. IEC 61499 FUNCTION BLOCK ONTOLOGY DEFINITION

A. Ontological Knowledge Base - Definitions and Examples

A typical knowledge base using ontology comprises two components: a T-Box and an A-Box [23]. T-Box stands for taxonomy box which describes concepts and their general properties. A-Box or assertion box retains knowledge that is specific to individuals or instances of concepts. In IEC 61499 terms, T-Box is the knowledge base of all properties and relationships between component-based software architecture concepts. All actually implemented component-based system configurations and function blocks are modeled in A-Box.

As described in the previous section, the library elements part is considered as the root node of the ontology. All sub-domains are defined under the major domain:

$$\text{Library Elements} \equiv \text{Common Elements} \sqcup \text{FB Types} \sqcup \text{Adapter Types} \sqcup \text{Resource Types} \sqcup \text{System Elements} \sqcup \text{Sub-Application Types} \sqcup \text{Network Elements.}$$ [1]

As no repeatable concept names is allowed in the ontology definition, the options are either to have all the ontology concepts of this component-based architecture named with the domain and sub-domain name (e.g. `<DomainName>_<SubDomainName>_<ConceptName>`) or store three root nodes in separate files without changing any concept name.

A concept in the architecture, or in another word – element, is linked to other elements via some roles. In terms of OWL, these elements are named as classes and these roles are called properties. There are two types of properties in the OWL ontology: Object Property and Data Property. An object property is used to describe a property value that refers to another object. Correspondingly, the data property is used when the property value refers to the actual data literal or a data type. In addition, extra information can be stored in annotation properties.

The object properties of this ontology model contain the hierarchy of the component-based architecture code structure for semantic analysis. When using an object property to represent an element requiring another element, the name of this object property is defined as *Has_<ConceptName>*. To complete this object property, domains and ranges are required. The domains are the classes where this object property will be used from and ranges are the classes where this object property will be applied to. An object property can be used in multiple locations in the same ontology model. Data properties are utilized to represent the attribute values of elements in the software architecture. When using a data property to present a constant value of a data type in the attributes of elements, the data property is named as *Has_<ConceptName>_<AttributeName>* or *Has_ConstantValue__<ConceptName>* when the element itself is a constant value. Similar to the object property, domain and range are required as well. In the data property, domains are the locations where this data property will be used from and ranges are the pre-defined data types in the XML Schema and OWL.

The idea of properties ontology definition will be illustrated on the concept of *FBType* of IEC 61499. The keyword *FBType* defines a function block type that can be basic, composite or service interface function block. The corresponding ontological definition comprises of Basic FB type or FB Network (Composite FB) or Service (Service Interface FB) element associated with an interface. Beyond those essential parts, there might be some extra details including compiler information, version information, etc. For the data properties,

[1] The notation definition of the description logic is listed in the Appendix.

a function block must have a name and may have some comments.

The first step of creating a class description is to create class itself with all data properties used in this class as well as related axioms. For instance, *FBType* must have a name of data type *String* (Characters). For example, it can be expressed as the following DL axiom:

$$=1\ Has\_FBType\_Name.String,$$

and then converted into OWL format (Fig. 2):

```
<rdfs:subClassOf>
  <owl:Restriction>
    <owl:onProperty rdf:resource="#Has_FBType_Name"/>
      <owl:qualifiedCardinality
          rdf:datatype="&xsd;nonNegativeInteger">1
      </owl:qualifiedCardinality>
    <owl:onDataRange rdf:resource="#String"/>
  </owl:Restriction>
</rdfs:subClassOf>
```

Fig. 2. Data Property Example of FBType.

The next step is to create all object properties for this class as well as related axioms. An *FBType* either has a Basic Function Block description or Function Block Network:

$$(\leq 1\ \text{Has\_FBNetwork.FBNetwork} \sqcup \leq 1\ \text{Has\_BasicFB.BasicFB}).$$

In OWL this class is presented in Fig. 3.

```
<rdfs:subClassOf>
  <owl:Class>
    <owl:unionOf rdf:parseType="Collection">
      <owl:Restriction>
        <owl:onProperty rdf:resource="#Has_BasicFB"/>
        <owl:onClass rdf:resource="#BasicFB"/>
        <owl:maxQualifiedCardinality
          rdf:datatype="&xsd;nonNegativeInteger">1
        </owl:maxQualifiedCardinality>
      </owl:Restriction>
      <owl:Restriction>
        <owl:onProperty rdf:resource="#Has_FBNetwork"/>
        <owl:onClass rdf:resource="#FBNetwork"/>
        <owl:maxQualifiedCardinality
          rdf:datatype="&xsd;nonNegativeInteger">1
        </owl:maxQualifiedCardinality>
      </owl:Restriction>
    </owl:unionOf>
  </owl:Class>
</rdfs:subClassOf>
```

Fig. 3 Object Property Example of FBType.

The overall ontology concept for *FBType* represented in Protégé tool [32] is listed in Fig. 4. The meaning of this description is that, an *FBType* individual must have exactly one interface, name, maximum one service, identification, version information, compiler information and comments, and maximum one Function Block Network or Basic Function Block description.

Fig. 4. *FBType* definition.

B. Automatic Generation of Ontological Knowledge Base

In order to quickly define the ontology T-Box of this component-based software architecture, an automatic converting methodology has been developed to reduce human errors during manual processes.

According to [24], a XML file schema can be converted to OWL automatically. A "DTD/XSD to OWL" transformation engine has been developed in this work for the automatic generation of ontology from XML Schema. The IEC 61499 XML format is specified in the form of three standard DTDs for Library Elements, Function Block Management Commands and Data types. Each DTD file is considered as a domain in the ontology and is converted and combined by the DTDtoOWL engine into a single ontological model. The mapping process is as follows:

1) Each DTD document is considered as a domain in the ontology. DTD Elements can be grouped into sub-domains if required.
2) Each DTD Element is mapped to an OWL class. In order to give a unique ID to each class, domain and sub-domain names must be added as prefixes for the class ID.
3) The hierarchies of the DTD Elements are mapped to the object properties and if the Element has only constants, they are mapped to the data properties straight away by using the prefix "*Has_ConstantValue_*". In a standard DTD document, there are some symbols to indicate occurrence of an element:

* Declaring Zero or More Occurrences of an Element;

\+ Declaring Minimum One Occurrence of an Element;

? Declaring Zero or One Occurrences of an Element;

The OWL keyword owl:QualifiedCardinality is used to represent those occurrence symbols, for example:

* Owl:minQualifiedCardinality = 0;

\+ Owl:minQualifiedCardinality = 1;

? Owl:maxQualifiedCardinality = 1;

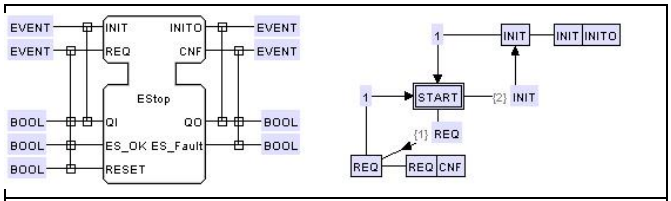
The attributes of an element are mapped to data properties. There are two sorts of attributes: #REQUIRED and #IMPLIED. The "Required" attribute is mapped to Owl:QualifiedCardinality with quantity of exactly one. The "Implied" attribute means that the attribute is not necessary to appear in the XML, which can be expressed by Owl:maxQualifiedCardinality = 1.

The DTD to OWL engine developed according to the rules above is able to generate the complete ontology T-Box of this component-based software architecture. The next step is to automatically import all function block files (*.fbt), resources

files (*.res), device files (*.dev) and system configuration files (*.sys) into the A-Box for semantic analysis.

An OWL individual is created for each XML element in the IEC 61499 source file. The object property *Has_<ConceptName>* is created for all child nodes of this XML element. Finally, data properties in the form *Has_<ConceptName>_<AttributeName>* are created for all attributes of this XML element with the actual value stored in them. Fig. 5 presents an interface and ECC of the basic function block "EStop" (that is a part of the example in Fig.1) which controls the valve's opening and closing based on the current position and the "start/end" position sensor readings. The ECC of the FB works as follows: at the input event REQ indicating a position change, the valve control command will be re-calculated and then the ECC returns to the idle state.

To describe this function block in the ontological knowledge base, an *owl:NamedIndividual* item is created in the form of *<NodeType>_<NodeName>* as *FBType_EStopZone*. To specify the node type of this instance, a rdf:type is inserted as the first sub-node with a type of *FBType*. All sub-elements, such as: *Identification*, *VersionInfo*, *CompilerInfo*, *InterfaceList* and *Basic FB* of *FBType* (refer to Fig. 5) are used to create object properties. Property *Has_<ConceptName>* is used to link to the *owl* individual of that particular sub-node. The attributes of this node are created as data properties. In our case, the data property *Has_FBType_Name* is created and refers to data type *String.* It indicates the name of this function block instance is *EStop.*




```
<owl:NamedIndividual rdf:about="#FBType_EStop">
   <rdf:type rdf:resource="#FBType" />
   <Has_Identification  rdf:resource="#FBType_EStop_Identification" />
   <Has_VersionInfo rdf:resource="#FBType_EStop_VersionInfo" />
   <Has_CompilerInfo rdf:resource="FBType_EStop_CompilerInfo" />
   <Has_InterfaceList rdf:resource="#FBType_EStop_InterfaceList" />
   <Has_BasicFB rdf:resource="#FBType_EStop_BasicFB" />
   <Has_FBType_Name rdf:datatype="STRING"> EStop</Has_FBType_Name>
</owl:NamedIndividual>
```

Fig. 5. Example of OWL Individual for the EStopZone FB type.

Following the rules presented in this section, all function blocks, resources, devices and system configurations can be imported directly into the ontology knowledge base A-Box.

VI. SEMANTIC ANALYSIS OF IEC 61499 ONTOLOGY MODEL

The ontological model of a particular artefact from the function block architecture (A-box) can be exposed to the semantic analysis. However, first the syntactic correctness of the artefact must be established. The generated DTD-based ontological model can be also used for that purpose. In the context of this work, a function block type is defined to be syntactically correct if all related object and data properties exist in the A-Box and the quantity constraints are also satisfied. However, there are also some contradictions in the syntax definition in the standard which need to be resolved prior to conducting the syntax check. In the following of this section, a part of semantic rules is defined.

The semantic analysis includes not only checking the system settings and parameters (for example, that all connections of FBs and mapping of variables are correct) or structural properties, but also the execution behaviors (for instance, determined execution order) [21] [26]. However, pure OWL DL is not sufficient to present such complex semantic rules, therefore SWRL is used here as the semantic rule language.

In order to interpret SWRL rules, a software tool called *reasoner* is required. The ontology query language SQWRL [22] is an extension of SWRL that allows more comprehensive representation of reasoning results similar to SQL queries. SQWRL takes a standard SWRL rule antecedent and treats it as a pattern specification for a query SQWRL has some essential extensions as against SWRL, for example, set and string operations.

Semantic rules represented in S(Q)WRL can be regarded as scripts written in a very high level programming language in order to search for a particular kind of semantic problem in applications complying with IEC 61499 standard. The ontological model can be compared to an API that provides access to basic artifacts of the program being analyzed. The benefits of the proposed approach are: 1) Compactness of the "scripts" – the search engine is built into S(Q)WRL implementation; 2) The "API" is generated automatically based on the syntax of input language; 3) Off-the-shelf tools (like Protégé) can be used as a semantic analysis platform.

In the remainder of this paper, the semantic rules are divided into various catalogues for each function block type and system configuration. SQWRL will be used for the rules definition due to a number of its powerful features as compared to SWRL. Some SQWRL rules are based on the description logic and can be expressed in the description logic format. A complete set of semantic rules was developed for IEC 61499 semantic analysis. The complete rule set consists of several subsets including rules for basic FBs, composite FBs, SIFBs, applications and system configurations. In this section, only the subset rules for basic and composite FBs are given as an example. It should also be noted the sources of the proposed rule set are the following: i) the text of the standard IEC 61499 (Part 1) [1], 2) the textual syntax of FB language (Annex B of Part 1 of the standard); 3) DTD for XML-documents representing FB systems (Annex A of Part 2 of the standard); 4) examples from FBDK.

Firstly, we define the semantic correctness of a basic FB as follows (this definition can be, of course, extended).

Definition 1. A basic function block type is said to be semantically correct if all of the following rules are satisfied:

(1) Inside the ECC, no identical EC transition condition is allowed from a single EC state;
(2) If an EC state connects to more than one EC state, always true "1" is only allowed in the lowest priority EC transition condition of those connected EC states;
(3) Each EC state must have at least one entry and one exit EC transition;

(4) Each event input is used in at least one EC transition condition;
(5) Each event output is used as at least one EC action output event;
(6) Each data input variable is associated with at least one event input of this basic FB;
(7) Each data output variable is associated with at least one event output of this basic FB;

The first three semantic rules ensure the execution semantic of ECC is correct. For example, the first rule states that if an EC state is linked to multiple EC states, no identical EC transition condition is permitted. Identical EC transition conditions can cause some EC states to be never reachable as ECC will always execute the EC transition with the highest priority among those identical EC transition conditions. This rule is presented as a SQWRL query below:

```
FBType(?FBType1) ^ Has_FBType_Name(?FBType1, ?name)
^ Has_BasicFB(?FBType1, ?BFB1) ^ Has_ECC(?BFB1, ?ECC1)
^ Has_ECState(?ECC1, ?ECState1) ^ Has_ECState_Name(?ECState1, ?name1)
^ Has_ECTransition(?ECC1, ?ECT1)
^ Has_ECTransition_Source(?ECT1, ?source1)
^ swrlb:stringEqualIgnoreCase(?source1,?name1)
^ Has_ECTransition_Condition(?ECT1, ?cond1)
°sqwrl:makeSet(?set1, ?cond1) ^ sqwrl:groupBy(?set1, ?name, ?name1)
^ sqwrl:makeBag(?bag1, ?cond1) ^ sqwrl:groupBy(?bag1, ?name, ?name1)
° sqwrl:notEqual(?set1, ?bag1) -> sqwrl:select(?name, ?name1)
```

In the SQWRL expression above, the left hand side is a SWRL rule antecedent. All operators starting with "sqwrl:" are the built-in functions from SQWRL. All EC Transitions with identical conditions grouped by EC transition source names will be listed by this SWRL rule. *Swrlb:stringEqualIgnoreCase* is a SWRL built-in function which compares two string value. *Sqwrl:makeSet* operator is used to construct sets of results without duplicate elements in the set. *Sqwrl:makeBag* operator is similar to the *Sqwrl:makeSet* operator but duplicate elements are allowed in the bag. *Sqwrl:notEqual* compares two sets/bags. To retrieve elements from a set/bag. The query prints FB types and EC states names where an error was detected.

The right hand side *sqwrl:select* will build a table using arguments as columns of the table. In this rule, the found EC transition condition instances, which satisfied all the properties' conditions, are stored in the list *?set1*. When an FBType instance satisfies all properties (for instance, *has_BasicFB*) as well as the built-in functions (for instance, compare two sets/bags), this instance is included into the search results.

The SQWRL expression of the second rule is as follows:

```
FBType(?FBType1) ^ Has_FBType_Name(?FBType1, ?name)
^ Has_BasicFB(?FBType1, ?BasicFB1) ^ Has_ECC(?BasicFB1,?ECC1)
^ Has_ECState(?ECC1, ?ECState1) ^ Has_ECState_Name(?ECState1, ?name1)
^ Has_ECTransition(?ECC, ?ECT1)
^ Has_ECTransition_Source (?ECT1, ?Source1)
^ swrlb:stringEqualIgnoreCase(?source1,?name1)
^ Has_ECTransition_Condition(?ECT1,?Cond1)
° sqwrl:makeBag(?bag1, ?cond1) ^ sqwrl:groupBy(?bag1, ?name, ?name1)
^ sqwrl:makeSet(?set1, 1) °  sqwrl:size(?bag_size, ?bag1)
^ swrlb:greaterThan(?bag_size, 1)
^ swrlb:subtract(?BagSizeSubtract1, ?bag_size, 1)
^ sqwrl:nth(?withoutLowestPriorityECTCond, ?bag1, 2, ? BagSizeSubtract1)
^ sqwrl:contains(?withoutLowestPriorityECTCond, ?set1)
-> sqwrl:select(?name, ?name1 )
```

Sqwrl:size will count the number of elements in a set or bag and stored in the variable indicated in the first operand. *Sqwrl:contains* check if a set/bag has all elements from another set/bag. *Swrlb:greaterThan* is a SWRL built-in function which compares its operands. *Sqwrl:element* is used to list all elements. In this semantic rule, all EC transitions with condition of always true having more than one EC transition are listed for correction.

The third rule is used to detect dead end of execution in the event chain. If an EC state has no output transition, once the execution of that function block instance is completed, it will stuck in that state forever and never execute anymore. The rule is given as two parts. The first part is for EC states without outgoing EC transitions:

```
FBType(?FBType1) ^ Has_FBType_Name(?FBType1, ?name)
^ Has_BasicFB(?FBType1, ?BasicFB1) ^ Has_ECC(?BasicFB1,?ECC1)
^ Has_ECState_Name(?ECState1, ?name1)
^ Has_ECTransition(?ECC1, ?ECT1)
^ Has_ECTransition_Source(?ECT1, ?source1)
°sqwrl:makeSet(?set1,?name1) ^ sqwrl:groupBy(?set1, ?name)
^ sqwrl:makeSet(?set2, ?source1) ^ sqwrl:groupBy(?set2, ?name)
° sqwrl:difference(?set, ?set1, ?set2)^ sqwrl:element(?e1, ?set)
-> sqwrl:select(?name, ?e1)
```

And second part is for EC states without incoming EC transitions:

```
FBType(?FBType1) ^ Has_FBType_Name(?FBType1, ?name)
^ Has_BasicFB(?FBType1, ?BasicFB1) ^ Has_ECC(?BasicFB1,?ECC1)
^ Has_ECState(?ECC1,?ECState1) ^ Has_ECState_Name(?ECState1, ?name1)
^ Has_ECTransition(?ECC1, ?ECT1)
^ Has_ECTransition_Destination(?ECT1, ?dest1)
°sqwrl:makeSet(?set1,?name1) ^ sqwrl:groupBy(?set1, ?name)
^ sqwrl:makeSet(?set2, ?dest1)^ sqwrl:groupBy(?set2, ?name)
° sqwrl:difference(?set, ?set1, ?set2) ^ sqwrl:element(?e1, ?set)
-> sqwrl:select(?name, ?e1)
```

This search requires an EC transition to have both source and destination. If a FB type is in the result list, either the EC transition source or destination of that FB type is missing.

The SQWRL implementation of the Rule 4 also uses set operations. The rule finds the set of input events which are not included in any EC transition. The rule's description is as follows:

```
FBType(?FBType1) ^ Has_InterfaceList(?FBType1, ?List1)
^ Has_EventInputs(?List1,?EI1) ^ Has_Event(?EI1, ?EV1)
^ Has_Event_Name(?EV1, ?EV1_Name)
^ Has_BasicFB(?FBType1, ?BasicFB1) ^ Has_ECC(?BasicFB1,?ECC1)
^ Has_ECTransition(?ECC1, ?ECT1),
^ Has_ECTransition_Condition(?ECT1, ?ECCond1)
^ swrlb:normalizeSpace(?ECCond1, ?ECCond1_Nospace)
^ swrlb:substringBefore(?ECCond1_Nospace, "&", ?EV2_Name)°
sqwrl:makeSet(?EV1Set, ? EV1_Name)
^ sqwrl:makeSet(?EV2Set, ?EV2_Name)
^ sqwrl:difference(?EVSet, ?EV1Set, ?EV2Set)
  -> sqwrl:select(?FBType1, ?EVSet)
```

The first part of the query will return all event input names of a single basic FB. If the event input name, being looked for, is in the result list, this event input is a part of this basic FB type interface. The second part is to gather all events used in the EC transition conditions. The event names are located before "&" symbol in EC transition conditions. Finally the set difference operator is used to find events that are in the first set but not in the second one. It should be noted that the above rule is intended for the situation when EC-transition condition consists of two parts – an event input and a guard condition (In

this case they are delimited by the sign "&") or only event input is presented. For other cases different rules are developed.

Other rules are similar to the rule 4, but applied to event outputs, and data inputs and outputs. For rule 6 and 7, it is syntactically correct that data inputs and outputs are not associated with any event inputs or outputs (allowed in the IEC 61499 standard). Those two rules just provide a warning to users there is a chance you may forget connecting those data inputs and outputs. Those rules could be applied to IEC 61499 IDEs to give a warning during compilation process.

Definition 2. A composite FB type is said to be semantically correct if all of the following rules are satisfied:

(1) The source and the destination data variable in a data connection must have consistent data types;
(2) Each event (data) input is connected to an event (data) input inside the included FB network or an event (data) output;
(3) Each event (data) output is connected from an event (data) output inside the included FB network or an event (data) input;
(4) Each data input variable is associated with at least one event input of this composite FB interface;
(5) Each data output variable is associated with at least one event output of this composite FB interface.
(6) No dangerous event loop is allowed.
(7) No short-Circuit event is allowed.

The rule 4 and 5 here are similar to the BFB rule 6 and 7. The semantic analysis just provides a warning as it is syntactically correct. An interesting part of a composite FB is the event connections in FB network. As defined in the rules 6 and 7, dangerous event loop or short event can cause unexpected behavior in the execution [25] [26].

Before start detecting dangerous event loops or short events, the event chain concept needs to be defined first. An event chain is defined as:

Definition 3. An event chain in a FB network is a sequence of event connections c_1, c_2,…,c_n which satisfies the following conditions:

i) Event connection c_i is connected to an event input of a FB instance *fb* and event connection c_{i+1} is connected from an event output of this FB instance *fb* (where $1 \leq i < n$);

ii) *fb* is "reactive" with respect to c_i and c_{i+1}, that means an event input from c_i triggers an event output to c_{i+1} via some manipulations inside *fb*.

An event chain is non-stoppable if $c_1 = c_n$ which forms an event cycle that keep looping indefinitely (like while(1) in C). A non-stoppable event chain is considered as a dangerous event loop if there is an external event input that initializes the event loop like a SIFB (Fig. 6 (A)).

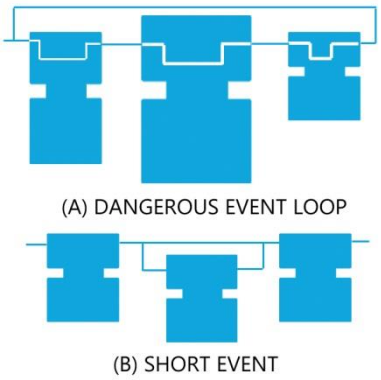


Fig. 6. Dangerous Event Loop and Short-Circuit Event.

The conditions of dangerous event loops are different for each FB *meta-type* and can be defined using the concept of causality. In a FB an event input ei_1 *triggers* an event output eo_1 if the following three conditions are true:

A. If FB Instance is Basic FB type

i) There is an EC transition and the event input ei_1 is the only EC transition condition attached for that EC transition;

ii) At the destination EC state of the EC transition there is an EC action firing the event output eo_1. A scheme for the event flow through a basic FB is shown in Fig. 7.

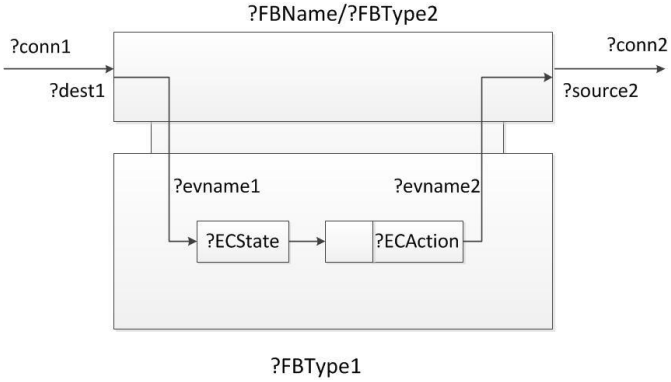


Fig. 7. Event flow through a basic FB.

Below an implementation of above concepts using S(Q)WRL rules is presented.

Here event chains are defined using relations (in other words, OWL object properties) *EvFlow* and *EvFlow_transitive* in SWRL. The event flow via a basic FB can be defined as the following SWRL rule. At that, parameter *conn1* of the rule consequent *EvFlow* represents a preceding event connection and parameter *conn2* represents a successive event connection in an event chain.

```
FBType(?FBType1) ^ Has_FBNetwork(?FBType1,?FBNetwork)
^ Has_EventConnections(?FBNetwork,?conns)
^ Has_Connection(?conns, ?conn1)
^ Has_Connection_Destination(?conn1, ?dest1)
^ Has_Connection(?conns, ?conn2)
^ Has_Connection_Source(?conn2, ?source2)
^ swrlb:SubstringBefore(?dest1, ".", ?FBName)
^ swrlb:SubstringBefore(?source2, ".", ?FBName)
^ swrlb:SubstringAfter(?dest1, ".", ?evname1)
^ swrlb:SubstringAfter(?source2, ".", ? evname2)
^ Has_FB(?FBNetwork,?FB)^ Has_FB_Name(?FB,?FBName)
^ Has_FB_Type(?FB,?FBType2) ^ Has_BasicFB(?FBtype2,?BFBType)
^ Has_ECC(?BFBType,?ECC)^ Has_ECTransition(?ECC,?ECTran)
^ Has_ECTransition_Condition(?ECTran,?evname1)
^ Has_ECTransition_Destination(?ECTran,?ECState)
^ Has_ECAction(?ECState,?ECAction)
```

^ Has_ECAction_Output(?ECAction,?evname2)
->EvFlow(?conn1,?conn2)

B. If FB Instance is SIFB type

There is a service, a service sequence, and a service transaction such that this service transaction has an input primitive triggered by the event input ei_1 and an output primitive fires the event output eo_1 (as illustrated in Fig. 8).

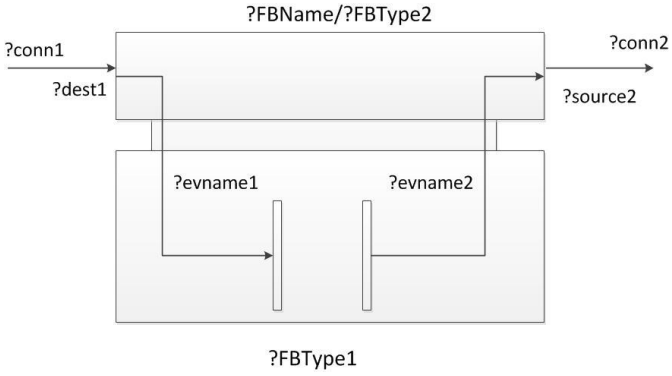


Fig. 8. Event flow through a SIFB.

The following SWRL rule expresses the event flow through SIFB.

FBType(?FBType1) ^ Has_FBNetwork(?FBType1,?FBNetwork)
^ Has_EventConnections(?FBNetwork,?conns)
^ Has_Connection(?conns, ?conn1)
^ Has_Connection_Destination(?conn1, ?dest1)
^ Has_Connection(?conns, ?conn2)
^ Has_Connection_Source(?conn2, ?source2)
^ swrlb:SubstringBefore(?dest1, ".", ?FBName)
^ swrlb:SubstringBefore(?source2, ".", ?FBName)
^ swrlb:SubstringAfter(?dest1, ".", ?evname1)
^ swrlb:SubstringAfter(?source2, ".", ?evname2)
^ Has_FB(?FBNetwork,?FB) ^ Has_FB_Name(?FB,?FBName)
^ Has_FB_Type(?FB,?FBType2) ^ Has_Service(?FBtype2, ?ser)
^ Has_ServiceSequence(?ser, ?ss) ^ Has_ServiceTransaction(?ss, ?st)
^ Has_InputPrimitive(?st, ?inprim)
^ Has_InputPrimitive_Event(?inprim, ?evname1)
^ Has_OutputPrimitive(?st, ?outprim)
^ Has_OutputPrimitive_Event(?outprim, ?evname2)
->EvFlow(?conn1,?conn2)

3. If FB Instance is Composite FB type

In FB network inside the composite FB there is an event chain beginning at the event input ei_1 and ended at the event output eo_1.

However, the relation *EvFlow* defined above only gives a pair of event connections via a function block. The following SWRL rules are used for computing *transitive closure* of relation *EvFlow* that defines an event chain recursively.

EvFlow (?conn1, ?conn2) -> EvFlow_Transitive(?conn1, ?conn2)
EvFlow (?conn1, ?conn2) ^ EvFlow_Transitive(?conn2, ?conn3)
-> EvFlow_Transitive(?conn1, ?conn3)

The SQWRL query below prints all event inputs and outputs of non-stoppable event chains of a flat FB network using the above SWRL rule for *EvFlow_Transitive*.

EvFlow_Transitive (?conn, ?conn)^ Has_Connection_Source(?conn, ?source)
^ Has_Connection_Destination(?conn, ?dest)-> sqwrl:select(?source, ?dest)

If the detected function block type is a composite function block, extra SWRL rule is required to find the event output from the nested function blocks with internal event connections.

The short event as illustrated in Fig 6 (B) is similar to the dangerous event loop rule. If an event is found bypassed a function block and that function block is non-stoppable, this will cause a semantic issue. The downstream function block will be invoked again after bypassing the event connection. Input data will be overwritten by the bypassed function block.

All the rules above ensure that components constituting an FB system are semantically correct. The execution semantics of those components is also required to be checked in the system configuration. An example of the semantic rule could be no indefinite event loop (deadlock) or short-circuit event to exist in system configurations. Another example can require every function block instance in the function block application to be mapped to a resource.

New semantic rules can be easily added to the semantic check system in the form of S(Q)WRL rules.

VII. Case Study of Baggage Handling System on Semantic Analysis

In this section, the defined semantic rules will be applied in a test case that is based on a component-based software control application for the airport inbound baggage handling system illustrated in Fig. 1. The application is designed using the function block tool FBDK.

The system consists of five transportation conveyors (IB101 to IB105), one inbound baggage carousel (IB1) and three emergency stop control station (ESTOP1 to ESTOP3). There is one electric photo eye sensor installed on the downstream end of each transportation conveyor to detect bags as well as on the carousel. Emergency stop control stations with reset push buttons are located around the inbound BHS system.

The FB system configuration is designed based on the original PLC version. A SIFB polling data inputs triggered by an E_CYCLE FB is used to represent a PLC scan. Original PLC function blocks for C*onveyor*, *EStop and EStopZone* are converted to their IEC 61499 version. Finally, data outputs are updated by another SIFB.

A semantic checker with OWL, SWRL libraries and SQWRL support is built for testing this case study. The complete rule set is implemented in this checker.

The analysis is applied to all FBs used in the system by the checker automatically: basic FBs C*onveyor*, *EStop and EStopZone*, and system configuration. The semantic checker goes through each rule defined for this FB type and generates warnings. First, the *Conveyor* FB is responsible for the basic conveyor control functionality. The *Conveyor* FB has a number of EC states and each state is representing an actual status of a conveyor. As shown in Fig. 9, the original ECC has two issues found by the semantic analysis. First, the RUN state has connections to the CASCADE, the ECM and the FAULT state. However, the transition condition to the CASCADE state is "1" and it is not the lowest priority transition. This will cause the conveyor to start and stop all the time, so the motor can be easily damaged in a short period of time. Secondly, the FAULT state has no exit EC transition. As a result, once any conveyor is faulted, it will remain in the fault state and cause a deadlock. Those issues are resolved in the corrected version of conveyor control ECC that passed the semantic check.

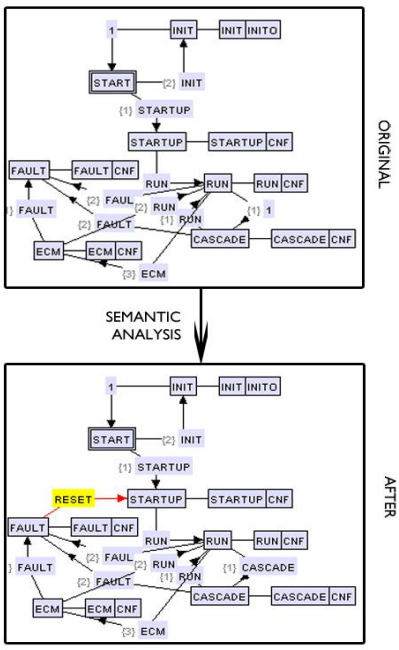


Fig. 9. Original and Corrected ECC of Basic Function Block Conveyor Control.

Through the semantic analysis for the BHS system configuration, a dangerous event loop is found as shown in Fig. 10. There is only REQ event condition and no EC guard condition for REQ state in the ECC of both the *EStop* function block and the *EStopZone* function block. Once the REQ input event of the function block *ESTOP1* is triggered, this indefinite event loop will be activated. The emergency stop functionality will not perform properly. This is a serious hazard as the system is not capable to ensure the safety requirements. The full path of this dangerous event loop is highlighted in the Fig. 10. This dangerous event loop is ignored by IEC 61499 compilers. To remove this dangerous event loop, the feedback event connection from EStopZone_All.CNF to ESTOP1.REQ is removed.

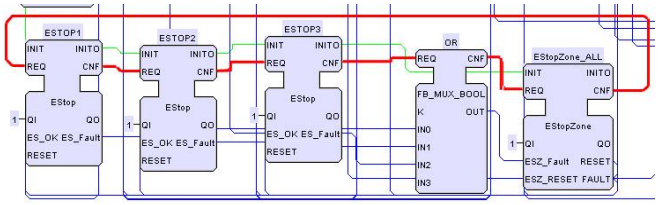


Fig. 10.Dangerous Event Loop Found in the Emergency Stop Control Part.

This approach is also scalable to more complex industrial implementations like process control systems. The semantic checker is capable for checking more nested function block structures and larger function block networks within a reasonable time automatically. Also if another rule needs to be added, it can be achieved by simply inserting a S(Q)WRL text into the semantic checker. The semantic checker will automatically load this new rule on startup.

VIII. IMPLEMENTATION

The semantic analysis of function block systems presented by their ontological models can be performed using standard ontological tools, such as Protégé [27] with the corresponding SWRL reasoners and SQWRL plugins.

A tool has been developed to generate automatically T-box ontological models from given DTD descriptions of syntax and import XML descriptions of function block systems into A-box as instances. The tool has also a built-in query engine for interpreting the complete SQWRL language and most part of the SWRL built-in libraries.

IX. CONCLUSIONS

This paper presented a multi-layered ontology based semantic analysis for component-based software architecture. The semantic analysis using ontological knowledge base can be helpful at all design stages of component-based software system if implemented in the corresponding integrated development environments.

The ontological knowledge base is generated from the IEC 61499 standard DTD files and all instances directly imported from XML files. This automates the process of knowledge base creation and simplifies implementation of the method in the future analysis tools.

The proposed approach has a number of advantages as compared to other approaches which develop dedicated validators for each semantic rule, as follows:

1) The formal method of Web-ontologies allowing to express precisely, clearly and at a high level many semantic properties of FB systems;

2) Formal representation of semantic properties allows usage of formal reasoning for the proof of these properties, that in turn increases reliability of the received outcomes of the analysis;

3) Usage of description logic (DL) and DL-safe SWRL rules [13] guarantees decidability of the task of classification (in our case - the analysis), that means the task of the analysis should be completed in comprehensible time.

4) The proposed method is flexible and extensible. Semantic analysis is performed by a universal engine that imports rules from extensible knowledge base.

The proposed method can be extended to semantic analysis of any visual programming language, especially based on XML notation.

The future work will be mainly focused on developing sets of semantic rules, for example, for specific design patterns or known semantic problems, as well as on optimization of the checker's performance. This will aim at analysis of deeper semantic issues in actual application logic inside FBs. Finally, it will be really useful if a linkage between problem domain and IEC 61499 ontologies can be established (for example, generate IEC 61499 ontologies from other ontologies describing system behaviours).

APPENDIX

Description Logic Notation Definition [23]:

Notation	Description	Example
$\sqcup$	Union of concepts	$C \sqcup D$
$\sqcap$	Intersection of concepts	$C \sqcap D$
$\equiv$	Concept equivalence	$C \equiv D$
$\sqsubseteq$	Concept inclusion	$C \sqsubseteq D$
$\exists$	Existential restriction	$\exists R.C$

¬	Complement	¬C

C and D are ontology concepts, R is the restriction condition.

ACKNOWLEDGEMENTS

This work is supported, in part, by the FRDF grant 3625072/9573 of the University of Auckland.

REFERENCES

[1] IEC 61499, Function Blocks, *International Standard*, International Electrotechnical Commission, Geneva, Switzerland, First Edition, 2005

[2] V. Vyatkin, "IEC 61499 as Enabler of Distributed and Intelligent Automation: State of the Art Review", *IEEE Transactions on Industrial Informatics*, Vol. 7, Issue 4, Page 768 – 781, 2011.

[3] IEC 61131-3, Programmable Logic controllers - Part 3: Programming languages, *International Standard*, Second Edition, 2003

[4] V. Vyatkin, "OOONEIDA: An Open, Object-Oriented Knowledge Economy for Intelligent Industrial Automation", *IEEE Transactions on Industrial Informatics*, Volume 1, No. 1, February 2006, Pages 4-17.

[5] Function Block Development Kit (FBDK). Holobloc Inc., 2008, Available: http://www.holobloc.org/

[6] ISaGRAF Workbench[Online], available from http://www.isagraf.com

[7] nxtControl GmbH, nxtControl - Next generation software for next generation customers [Online, 2009, June]. Available: http://www.nxtcontrol.com/

[8] DTD Attributes[Online], W3C School Tutorial, retrieved from http://www.w3schools.com/dtd/

[9] C. Welty, "Semantic Web Ontologies", IBM Research[Online], http://www.daml.org/meetings/2005/04/pi/Ontologies.pdf

[10] V. Dubinin, V. Vytakin, "Ontology of IEC 61499 function blocks", *International conference on Contemporary information technologies (CIT'10)*, Penza, December, 2010.

[11] J. Lastra, I. Delamer, "Semantic Web Services in Factory Automation: Fundamental Insights and Research Roadmap", *IEEE Transactions on Industrial Informatics*, Vol 2, No. 1, Feburary 2006, Page 1-11.

[12] Y. Al-Safi, V. Vyatkin, "Ontology-based Reconfiguration Agent for Intelligent Mechatronic Systems in Flexible Manufacturing", *International Journal of Robotics and Computer Integrated Manufacturing*, Vol 26, Page 381 - 391, 2010

[13] B. Motik, U. Sattler, R. Studer. "Query answering for OWL DL with rules", *Journal of Web Semantics*, 3(1):41-60, 2005.

[14] SWRL: A Semantic Web Rule Language Combining OWL and RuleML[Online], retrieved from http://www.w3g.org/Submission/SWRL/

[15] Arakawa N. Semantic Analysis based on Ontologies with Semantic Web Standards, Int. Conf. Computer-aided Acquisition of Semantic Knowledge (CASK-2008), Sorbonne, 2008.

[16] Ontologies for Software Engineering and Software Technology / Calero, Coral; Ruiz, Francisco; Piattini, Mario (Eds.), 2006, XIV, 339 p., Springer.

[17] F. Jammes, H, Smit, "Service-Oriented Paradigms in Industrial Automation", *IEEE Trans Industrial Informatics*, 1(1), 2005, pp.62–70

[18] K.M. Goh, W. Dint, "Iterative Knowledge Based Code Generator for IEC 61499 Function Block", *2009 IEEE Region 10 Conference (TENCON 2009)*, Singapore, 23-26 Jan 2009, ISBN:978-1-4244-4546.

[19] D. Gasevic, D. Djuric, V. Devedzic, "Model Driven Engineering and Ontology Development", 2nd ed., Springer, Berlin Heidelberg New York, 2009, XXI, 378 P. (ISBN: 978-3-642-00281-6)

[20] P. Novak, R. Sindelar, "Application of Ontologies for Assembling Simulation Models of Industrial Systems", Proceedings of the 2011 *International Conference on On the move to meaningful internet systems*, 2011.

[21] V. Vyatkin, "The IEC 61499 Standard and its Semantics" – *IEEE Industrial Electronics Magazine*, Vol. 3, Issue 4, Page 40 - 48, 2009.

[22] M. O'Connor, A. Das, "SQWRL: a Query Language for OWL", *OWL: Experiences and Directions (OWLED), Fifth International Workshop 2009*, Vol 529.

[23] F. Badder, D. Calavanese, D.L. McGuinness, D. Nardi and P.F. Patel-Schneider, "The Description Logic Handbook, Theory, Implementation and Applications, 2nd Edition.", *Published by Cambridge University Press*, 2007, ISBN 978-0-521-87265-4

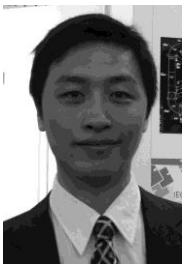
[24] P. Thuy, Y. Lee, S. Lee, "DTD2OWL: Automatic Transforming XML Documents into OWL Ontology", *2nd International Conference on Interaction Sciences: Information Technology, Culture and Human*, 16–18 Aug 2009, ISBN: 978-1-60558-710-3

[25] G. Cengic, K. Akesson, "On Formal Analysis of IEC 61499 Applications, Part A: Modeling", *IEEE Transactions on Industrial Informatics*, Volume 6, Issue 2, 2010, Page 136 – 144.

[26] G. Cengic, K. Akesson, "On Formal Analysis of IEC 61499 Applications, Part B: Execution Semantics", *IEEE Transactions on Industrial Informatics*, Volume 6, Issue 2, 2010, Page 145 – 154.

[27] Protégé. Available at http://protege.stanford.edu

[28] O. Orozco, J. Lastra, "Adding Function Blocks of IEC 61499 Semantic Description to Automation Objects", *IEEE International Conference on Emerging Technologies and Factory Automation*, Prague, Czech Republic, September, 2006, pp 537 – 544.



Wenbin Dai (GM' 09, M' 13) received a Bachelor of Engineering (with honours) degree in Computer Systems Engineering from the University of Auckland, New Zealand in 2006. He completed PhD in Electrical and Electronic Engineering at the Department of Electrical and Computer Engineering, The University of Auckland, New Zealand in 2012. His research interests are IEC 61131-3 PLC, IEC 61499 function blocks, distributed control systems, industrial fieldbus communication protocol, SOA and Internet of Things in industrial automation.

He has been also a software engineer from Glidepath Limited – a New Zealand based airport baggage handling system provider since 2007. He has involved in numbers of airport baggage handling system and parcel sortation system projects in New Zealand, Australia, Canada, China, Africa, Middle-East and South America. His responsibility is to design and develop PLC control and SCADA/HMI for those systems.

Valeriy Vyatkin is Chaired Professor of Dependable Computation and Communication Systems at Luleå University of Technology, Sweden, and visiting scholar at Cambridge University, U.K., on leave from The University of 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. He graduated with the Engineer degree in applied mathematics in 1988 from Taganrog State University of Radio Engineering (TSURE), Taganrog, Russia. Later he received the Ph.D. (1992) and Dr. Sci. degree (1998) from the same university, and the Dr. Eng. Degree from Nagoya Institute of Technology, Nagoya, Japan, in 1999. His previous faculty positions were with Martin Luther University of Halle-Wittenberg in Germany (Senior researcher and lecturer, 1999–2004), and with TSURE (Associate Professor, Professor, 1991–2002).

Research interests of professor Vyatkin are in the area of dependable distributed automation and industrial informatics, including software engineering for industrial automation systems, distributed architectures and multi-agent systems applied in various industry sectors: Smart Grid, material handling, building management systems, reconfigurable manufacturing, etc. Dr Vyatkin 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. In 2012, Prof Vyatkin has been awarded Andrew P. Sage Award for best IEEE Transactions paper.