Towards Industrially Usable Agent Technology for Smart Grid Automation

Gulnara Zhabelova, *Non Member*, Valeriy Vyatkin, *Member, IEEE* and Victor Dubinin, *Non Member*

***Abstract*—This work is aimed at facilitating industrial adoption of agent technology. The paper proposes the hybrid agent architecture specific to the power system automation domain. The architecture builds on the Logical Node concept of IEC 61850 and comprises of a deliberative and a reactive layers; combining advantages of both. By relying on the underlined industrial standards IEC 61850 and IEC 61499, the architecture ensures practical applicability and captures domain specific concepts in the agent-based system design. Developed agent-based system was validated in the co-simulation framework. The architecture provides for rapid system development, reducing the software development life cycle. The benefits are in trace-ability of the software requirements, re-use of software components, ease of re-design, and direct deployment of the system model.**

***Index Terms*—Agent architecture, Smart Grid, reactive and deliberative architectures, IEC 61499, IEC 61850, deliberation, power system automation, distributed grid intelligence.**

I. INTRODUCTION

With the introduction of widespread distributed energy generation, which motivates bi-directional flow of energy, current electrical grid needs to evolve from *energy delivery network* into an *energy exchange network* i.e. *Energy Internet*.

To control such a complex and highly distributed infrastructure, the Smart Grid has to employ new generation distributed automation and control systems, i.e. Distributed Grid Intelligence (DGI). DGI is a network of distributed nodes performing intelligent control to achieve local goals and participating in overall Smart Grid operation and control to achieve system objectives. These nodes are essentially agents operating autonomously, reacting on the environment and proactively negotiating among themselves to realize system objectives, exhibiting social behavior. DGI of Smart Grid is best realized as a distributed multi-agent system (MAS).

An agent in a broader sense is a system (as an entity) that can perform autonomous actions based on its interaction with the environment and other agents [1]. The central notion of agency is "autonomy" in making decisions and achieving delegated goals [1, 2]. An *intelligent* agent exhibits the following capabilities: reactivity (ability to perceive environment and respond in timely fashion), "proactiveness" (goal-directed behaviour, taking initiative to meet assigned objectives) and social ability (negotiate and cooperate with other agents) [1].

However, agent technology in the power system domain is the realm of theory and laboratory simulation [3]. There is a large gap between laboratory setups and industrial field systems. Agent-based systems need to be more practical in order to be applied in the power system automation domain and begin adapting to the specific requirements.

This work is aimed at facilitating industrial adoption of agent technology. The proposed practical agent architecture reflects concepts of power system automation domain and based on industrial standards IEC 61499 and IEC 61850.

IEC 61850 is a well-accepted standard in industry. It standardises representation of the automation functions and communication protocol for power system automation. IEC 61499 is open reference architecture for developing distributed systems in automation and control domain. As IEC 61499 addresses challenges of distributed systems design, the proposed agent architecture inherits advantages gained by IEC 61499, such as ability to deploy agents on the standard-compliant field devices.

The rest of the paper is structured as follows. Section II provides an introduction into the agent architectures and a discussion on the existing industrial agent architectures. Section III presents proposed automation agent architecture. Running examples of its implementation is provided. The proposed architecture is implemented and described in section IV on the example of distributed Fault Location Isolation and Supply Restoration (FLISR) application. This section also presents results and discussion on the proposed design of developed agent-based system. The section gives a comparative analysis of the proposed agent design with the existing FLISR agent based implementation. The paper ends with concluding remarks in section V and the future work.

Manuscript received April 1, 2014; revised July 5, 2014 and August 22, 2014; accepted September 28, 2014

This work has been supported, in part, by FREEDM NSF Centre grant Y3.E.C12

Gulnara Zhabelova is with Department of Computer Science, Computer and Space Engineering, Lulea Tekniska Universitet, 971 87 Lulea, Sweden. (email: Gulnara.zhabelova@ltu.se).

Valeriy Vyatkin is with Department of Computer Science, Computer and Space Engineering, Lulea Tekniska Universitet, 971 87 Lulea, Sweden and Department of Electrical Engineering and Automation, Aalto University, Finland.

Victor Dubinin is with Department of Computer Science, University of Penza, Russian Federation.(email: victor_n_dubinin@yahoo.com)

II. AGENTS IN POWER SYSTEM APPLICATIONS

A. Agent architectures

Wooldridge in [1] presents a general idea of an agent as a computer program situated in the environment and capable of autonomous actions within this environment and able to initiate those actions to achieve delegated objectives. An agent will perceive the environment through sensors, evaluate the current state, make a decision about the action to take, and act upon the environment through a set of available actions. An action can have associated pre-conditions. The key problem is the decision of which action from an available set of actions an agent should perform to best satisfy the design objectives.

An *agent architecture* is a "software architecture for decision making systems that are embodied in an environment" [1]. More definitions of agent architecture can be found in [4,5]. There are three major agent architectures in agent theory identified by Wooldridge and Jennings in [1]: reactive, deliberative and hybrid.

Reactive agents base their decisions on the present, without reference to history; they simply respond to their environment [1, 2]. The reactive architecture directly links sensory data to acting capabilities of the agent [4]. That is, the action to be performed is specified upon possible sensory input. During operation, the agent needs to continuously read the inputs and check against the conditions of each action. However, it is difficult to implement pro-activeness and goal-oriented behaviour based solely on reactive architecture. This architecture only looks at the present situation and does not take into account long term goals.

In the deliberative architecture the goals and plans are explicitly represented. The most popular deliberative agent architecture is the Beliefs, Desires and Intentions (BDI) of Rao and Georgeff [1]. The beliefs are the sensory input accumulated over time; desires are the delegated goals. Based on its beliefs and desires, an agent decides on its intentions, i.e. what it wants to do in the future. The intentions define the actions the agent is committed to perform. However, if the situation has changed, the agent can abandon the intention.

The disadvantage of this architecture is the time needed for an agent to evaluate the updated beliefs and form an intention. For complex agents it may take a relatively long time, and for highly dynamic environments it may be too long to react to changes.

Thus, many researchers agree that the combination of classical deliberative and alternative reactive architectures is a suitable approach for building agents [1,4].

A hybrid agent combines both reactive and deliberative architectures usually by introducing a layer of each into one agent [1,4]. Hybrid architectures implement autonomous reactive and pro-active agents. Often thanks to the co-operation layer, it can exhibit social behaviour. For instance, InteRRaP architecture is proposed by Muller [1,4]. In this architecture, the sensory data are directly linked to the behaviour layer, where immediate actions are specified. If the input data does not find a match in the behaviour layer, it is passed to the next plan layer, and so on to the higher layers of the architecture. Whichever layer has chosen the action, it will pass the action down to the lowest layer, which has to execute the corresponding action.

A disadvantage of this architecture is the complexity of coordination of different layers in order to achieve logical and consistent agent behaviour [4]. There are no formally defined semantics or methodology for developing agents of such architecture [4].

B. Agents in an industrial environment

It is a challenge to develop practical agent architecture, such that the agent would be executable on field devices and able to operate within industrial environment. In industrial settings there are tasks, which demand reaction under strict timing requirements; that limits the amount of possible computations. Traditional agent reasoning such as deductive and practical reasoning, used in the classical agent theory, and others employed in the Artificial Intelligence, are computationally intensive tasks and require symbolic representation of the agent's perception, over which they perform the deliberation process and means-end reasoning. These processes are expensive in terms of the required resources and time. Whereas, industrial agents must operate under time and resource constraints, such as fixed size memory and fixed processor performance. Additionally, the agent should be of reasonable complexity so that it is practical for a domain engineer to design, implement and execute agent-based systems on field devices.

Several attempts have been made to address this issue in the domain of automation of manufacturing systems. The works of Rockwell Automation [6], Ferreira [7], Ulewicz and Vogel-Heuser [8], Hegny [9] and Lepuschitz [10] have proposed agent architectures targeting execution platforms such as Programmable Logic Controllers (PLCs).

The complexity and the resource demand of the deliberation process, have lead to the solution of splitting the agents into two parts and two physical platforms. The decision making algorithms are implemented on high level languages such as Java, C# or C++ and executed on general purpose computers (PCs). And the other part performing time sensitive tasks and interfacing to I/O (actuators and sensors) are executed on the PLCs. A single agent is developed using several technologies such as Java and IEC 61131-3.

In the works [6-10] the agent architecture has two parts: High Level Controller (HLC) and Low Level Controller (LLC). HLC represent decision making part of the agent, whereas LLC executes commands from HLC and also performs direct control of actuators. HLC is often implemented in JADE, where HLC is designed as a complete agent [7,9]. HLC MAS is executed on the PC [7-10]. LLC is implemented on PLC with IEC 61131-3 languages [7-8] or IEC 61499 [9-10]. The interface between HLC MAS (JADE or C#) and PLCs is realized with Automation Device

Specification (ADS). ADS allows for high level programs to access PLC code and I/O. Alternatively, Rockwell Automation has implemented HLC in C++ and embedded it into PLC with the corresponding LLC [6]. In this way an agent is executed on a single platform. However, the PLC firmware had to be customised for this project [6].

The mentioned works tend to separate the resource and time demanding decision making process from time-critical reactive behaviour of an agent. Additionally, in these agents HLC passes down the commands to LLC to act on the environment. This type of agent architecture can be categorised as vertical layered architecture, according to the agent's classification introduced in [1].

The challenge is still remaining, i.e. suitable architecture enabling an agent to perform on field devices and operate within industrial environment. Furthermore, separation of the architecture over several technologies introduces development, integration and deployment challenges, and may incur runtime inefficiencies.

The aim of this paper is to propose an architecture integrating both deliberative and low level control within a single entity yet executable on field devices. The next chapter will describe the proposed agent architecture. Since IEC 61850 and IEC 61499 are well documented and widely used in research projects, their description is easily accessible in various publications. Therefore, due to the paper size limitation, the description of both standards will be left out.

III. Architecture of an Automation Agent

The agent architecture is based on the IEC 61850 and is realised using the IEC 61499 reference architecture.

The IEC 61850 standard "Communication networks and systems for power system utility automation" [11] reflects the advanced industrial practice for designing substation automation, control and protection applications. IEC 61850 decomposes the power substation down to objects so-called Logical Nodes (LN). A LN describes an information model of a domain specific automation function. For example, a function "overcurrent protection" is modelled as PIOC LN. The standard has a complete domain information model containing classes, their properties, taxonomy and relations between classes. IEC keeps improving and expanding the standard adding more information about the domain. Since the application domain of the proposed agent architecture is in power system automation, the IEC 61850 information model provides a perfect foundation for the agent.

IEC 61499 is designed for development of distributed systems in automation and control. The IEC 61499 standard "defines an open reference architecture for next generation of distributed control and automation" [12]. The design structures used in the standard are Basic Function Block (FB), Composite FB and Service Interface FB (SIFB); Application and abstract Device Model. FB encapsulates automation functions or any programming modules in a portable and re-usable platform independent form. The event-driven semantics of IEC 61499 can provide faster reaction time than the cycle-based execution of industrial controllers, such as PLCs [12].

A. Overview of the proposed architecture

In the proposed architecture, the agent models LN and its functionality is determined by the LN's objectives and functions. The IEC 61850 decomposition of the domain into automation functions gives a well-defined and structured domain model and helps with the agent identification. The agent inherits the information model of the LN as its beliefs and the domain specific function of a LN as its desires or objectives. The beliefs and the desires will define the intentions of the agent. Therefore the LN determines actions of the agent. The resultant set of agents is specific to the power system automation domain, reflecting given functional and non-functional requirements.

IEC 61850 communication standardize the information exchange within the power system automation schemes, however, it is not sufficient for agent communication.

At this stage we do not propose to use IEC 61850 communication services for inter-agent communication.

In order to serve the needs of communicating agents, the semantics of the messages need to be defined. One possibility is to consider FIPA developed Agent Communication Language (ACL) with SL, describing semantics of the messages. Another possible method is to use common knowledge representation, such as ontology.

The applicability of these approaches needs to be investigated in the future work. Agent communication language is out of scope of this paper.

Let us denote a Logical Node as *ln*, a set of beliefs as *B*, a set of desires as *D*, and *Action* as a set of possible actions of the agent. A logical node can then be defined as a tuple, containing the agent's beliefs and desires, $ln = (B, D)$. Let *LN* be a set of such logical nodes, $LN = \{ln_1, ln_2, \dots, ln_n\}$. Then the agent will perform actions defined by the respective logical node, $agent: LN \rightarrow Action$.

The proposed agent has a hybrid architecture (according to the agent's classification introduced in [1]), combining deliberative and reactive layers, and has a horizontal layered architecture, as depicted in Fig. 1. In the horizontal architecture, the sensory inputs are provided to each layer, and each layer generates its own actions. The architecture is made up of a reactive layer, a deliberative layer and beliefs.

The abstract model of the layered architecture with IEC 61499 FB is shown in Fig. **2**. The LN defines intentions, possible actions and dictates the structure of beliefs. The beliefs of the agent are IEC 61850 defined LN data. There is a library of modelled plans. A plan is implemented as a basic FB. The library of possible goals, or intentions, can be expressed as variables and stored within the plans as post-conditions or in the interpreter as pre-compiled options for the deliberation process. The interpreter is a separate FB, which can process beliefs, intentions and plans, and decide on a single plan to execute.

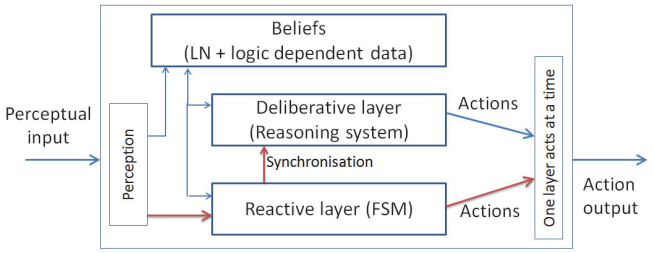

Fig. 1. Horizontally layered architecture of the proposed agent.

The reactive layer is represented by *Reactive FB* and can take precedence over the deliberative layer to perform its actions. The reactive *behaviour* is modelled as the IEC 61499 Execution Control Chart (ECC).

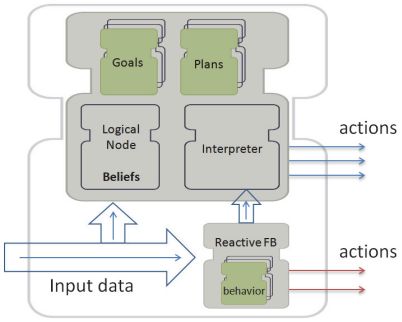

Fig. 2. Abstract FB model of the proposed horizontally layered agent architecture.

The next section describes reactive layer in detail.

B. *Reactive layer of the proposed arcthiecture*

In the proposed agent architecture, the reactive layer has the highest priority to ensure mission critical tasks are performed in a timely manner. The subsumption architecture of the reactive layer has first been proposed by Brooks [1]. It is built on a set of *task-accomplishing behaviours* [1]. Subsumption is a categorisation of the behaviours, where the higher layers represent more abstract behaviours. The *behaviour* is a function, which selects the action to be performed based on perception and current beliefs of the agent. The *behaviour* modules are arranged into a *subsumption hierarchy*, with the lowest layer (*behaviour*) having the highest priority. Fig. 3 shows the concept of the subsumption architecture [1]. Each layer is a *behaviour* module, which takes perceptual inputs and maps them to a suitable action. There might be several actions firing at once, therefore there needs to be a mechanism (arbitrator) to select the action with the highest priority.

In the proposed architecture, *behaviour* is modelled as IEC 61499 ECC, which is a form of FSM. The ECC defines the reaction of the basic FB on input events and data, thus mapping inputs into the appropriate action. An action in the ECC can be associated with an algorithm and output events. An algorithm operates over input, output and internal data. Output events are emitted once an algorithm is executed. An ECC is realized within a basic FB. Therefore, the *behaviour* component of the subsumption architecture is modelled by a basic FB. Definition of the basic FB formulates the description of the *behaviour* module of the agent. The formal model of IEC 61499 has been proposed by a number of researchers [13-18]. The proposed agent architecture is based on the IEC 61499 formal definition by Dubinin and Vyatkin [18]. The full formal model of IEC 61499 can be found in [18].

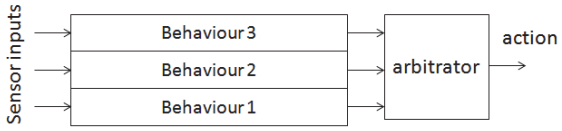

Fig. 3. Behaviour layers of subsumption architecture. An action selection [1].

A set of task accomplishing *behaviours* (basic FBs), can be represented as composite FBs. Formal model of a composite FB can be found in [18].

In the proposed architecture, priority is assigned to the *behaviours* if the actions of these behaviours overlap and compete. If the fired actions do not compete, then they are allowed to act simultaneously. Competing actions have to be arbitrated. Arbitration mechanism is left up to the designer and its agent specific (function). One method of arbitration is to group competing actions and resolve them with an arbitrator, which takes priority of the behaviours into account. Therefore, there will be one arbitrator per group of competing actions.

Fig. 4 depicts an example of reactive behaviour of the CSWI agent. CSWI agent has 2 reactive behaviours: *OperateXCBR* and *FaultIsolatedKeeper*. The first one operates the local circuit breaker on requests from the deliberative layer and requests from other CSWI agents. *FaultIsolatedKeeper* records the asynchronous event of the fault isolation, coming from the deliberative layer or other CSWI agents. These are implemented as ECC within basic FBs. ECC of the *OperateXCBR* reactive behaviour is a simple FSM (Fig. 4). On request from either the deliberative layer or the another agent, the desired operation on local circuit breaker is sent to XCBR agent. On the confirmation of the switch operation, the CSWI agent acknowledges the operation to the caller and the ECC returns to the original state.

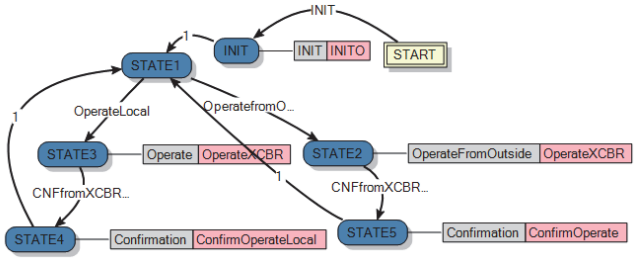

Fig. 4. OperateXCBR behaviour modelled as ECC of basic FB.

The *Behaviour* modules of CSWI reactive layer are not competing, therefore in this instance an arbitrator is not required.

As seen from the example, the *behaviours* contain simple actions. Simple ECC is easy to understand, debug and verify.

The reactive layer is responsible for situations, which need immediate attention. However, the agent may also have long- term goals. The deliberation process is captured in the deliberative layer of the proposed architecture. The method of deliberation can take different forms. This is discussed in the next section.

C. Deliberative layer of the proposed architecture

Deliberation or reasoning process is usually computationally intensive in terms of both time and memory. An industrial agent operates under resource constraints of field devices such as fixed size memory and fixed processor performance. These limitations restrict the amount of computation the agent can perform. Thus the reasoning should be effective and decisive. One such method is *Procedural Reasoning System* (PRS) [1]. PRS implements the BDI architecture and provides the structures of the BDI concept [19].

The PRS is the basis of the deliberative part of the proposed architecture.

The reasoning process involves two parts: deliberation and means-end reasoning. Deliberation resolves what state of affairs the agent wants to achieve. Means-end reasoning is a planning process, deciding how to achieve the desired state of affairs. The agent, through the deliberation process, will generate intentions, and by means-end reasoning perform planning and arrive to a plan of execution.

The reasoning engine is referred to as the *interpreter*. The interpreter will consider beliefs, desires, possible plans and current intentions in the deliberation process to select a plan, which will realize the desired state of affairs.

In procedural reasoning, there is no planning in principle. Besides, the actions of an industrial agent are known and restricted at the design time.

By establishing the agent on the basis of well-defined LN, the purpose of the agent, its desires, intentions and possible actions are known at the design stage. This allows designing a library of possible and sound plans. The pre-compiled plans can simplify the deliberation process and remove means-end reasoning. The option generation function is simplified to a match and select, rather than generating options/goals from scratch. The filter function is stripped down to the selection of the suitable plan comparing their utility ratings, priorities or using a meta-level plan. This results in agent decision making being simpler, faster and less demanding on resources.

Since the proposed agent is based on an LN, the LN defines the agent's beliefs and its purpose and intentions. The domain specific function of the LN defines set of possible actions of the agent. Therefore, the intentions and all possible plans can be provided at compile time as a library.

In this manner, the required processing resources can be reduced. Firstly, by an abridged deliberation process, which is still sufficient for intention filter function; and secondly, by substituting means-end reasoning with a library of pre-compiled plans.

A derived formal model of the proposed deliberative layer is presented in [20]. A short summary of the deliberative layer is described below.

Fig. 5 presents control loop of the proposed deliberative model. Let B_ln be current beliefs of the agent and Bel_ln be a set of agent's beliefs, then $B\_ln \subseteq Bel\_ln$. The desire D_ln is pushed into the intention stack at the start up. This will motivate the agent to operate and generate new intentions. The rest of the algorithm is the loop. The loop starts updating the beliefs with the perceptual inputs with function $see: E \rightarrow Per$, where *Per* is a set of perceptions and *E* is a set of states of the environment. The intention to be achieved is always at the top of the stack. The agent gets the intention I_ln from the stack, and decides how to achieve it. This will result in selection of a plan π to commit to. This plan is then executed. The procedure *execute(α)* performs a given action $\alpha \in \pi$. The action α may push new intentions into the stack. A plan in this architecture is a sequence of actions $\pi = \alpha_1, \alpha_2, \dots, \alpha_n$.

```
1. B_ln ← B_0_ln;                    /* B_0_ln are initial beliefs */
2. Stack ← D_ln;                     /* D_ln is the top level goal or desire*/
3. while true do
4.    Per ← see(E);                  /* Get the next percept Per */
5.    B_ln← brf (B_ln,Per);          /*Update beliefs*/
6.    I_ln ← hd(Stack);              /*Copy intention at head of Stack into I_ln*/
8.    P ← options(Plans_ln,I_ln,B_ln);
9.    π ← filter(P,N);
10.   while not (empty(π) or succeeded(I_ln,B_ln,π) or impossible(I_ln,B_ln,π)) do
11.         α ← head(π);
12.         execute(α);
13.         π  ← tailPlan(π);
14.         Per ← see(E);            /*Get the next percept Per */
15.         B_ln ← brf(B_ln,Per);    /*Update beliefs*/
16.         I_ln ← hd(Stack);        /*Copy intention at head of Stack into I_ln*/
17.         if (reconsider(I_ln,B_ln,π)) then
18.               P ← options(Plans_ln,I_ln,B_ln);
19.               π ← filter(P,N);
20.         end_if;
21.   end_while;
22    Stack ← tail(Stack); /*Remove current intention from head of stack */
23. end_while;
```

Fig. 5. Control loop of the proposed deliberative agent.

The deliberation process is captured in functions of option generation $options: 2^{Plans\_ln} \times Int\_ln \times Bel\_ln \rightarrow 2^P$ and filter $filter: 2^P \times \mathbb{N} \rightarrow P$, where Int_ln as a set of intentions, $I\_ln \in Int\_ln$ and $Plan\_ln = \{\pi_1, \pi_2, \dots, \pi_n\}$ is a set of all plans designed for the agent (a library of plans). An option generation function selects the possible plans to achieve, given intention and current beliefs. This set of possible plans capable to achieve current intention is denoted as $P, P \subseteq Plans\_ln$. From this generated set of possible plans P (options), a filtering function selects one plan π_i according to the defined criteria. Let us assume, that the plan is selected based on utility rating, which is expressed with a number from a set of natural numbers $\mathbb{N}$.

The agent implements *single-minded commitment* with *while* loop in rows 10-21. The *while* loop is dedicated to a chosen plan and will continue until there are no actions in the plan to execute or the intention has been succeeded or become impossible. The agent is reasonably *open-minded*.

The agent maintains an intention as long as it is still believed necessary to achieve. The $reconsider(I\_ln, B\_ln, \pi)$ function (row 17) checks if the intention, the current plan is aimed for, is still valid, i.e. there is still a reason to achieve the intention. If it decides the current intention (lets denote it as I_j_ln) should be reconsidered, an agent generates new options and filters out a new plan. And the *while* loop 10-21 will start executing the new plan.

CSWI agent is depicted on Fig. 7. Deliberative layer of the CSWI agent has five plans: *ProvideAltertnavieSupply*, *FaultIsolation*, *FaultLocated*, *Restore* and *GetHelp*. There is an *IntentionStack* and the reasoning engine - *Interpreter*.

The interface of the *IntentionStack* FB allows for plans to push new intentions and the *Interpreter* to read the top of the stack and to delete an intention when necessary. A plan is implemented with a basic FB. Each plan has input *StartPlan* and outputs *PlanIsCompleted* and *newInt* associated with *newIntVar*. Using the provided interface, the plans can push new intentions into the *CSWI_Stack*. Each plan also has a logic dependent interface for reading current beliefs and outputing actions. The interface of the *Interpreter* provides signals for controlling stack and plans and also reads necessary beliefs in order to deliberate a suitable intention and a plan.

The behaviour of the deliberative layer is controlled by the *Interpreter*, depicted in Fig. 6. The *Interpreter* requests the intention at the top of the stack and starts deliberating. There are three defined intentions for CSWI agent: "*NormalOperation*", "*FaultOperation*" and "*Restore*". The *Interpreter* performs one plan at a time. Once plan is completed, the *Interpreter* updates the intention and starts the deliberation considering the updated beliefs.

Formalised deliberative and reactive layers compose the proposed hybrid agent architecture. These two layers have to work together and synchronize their actions, which are described next.

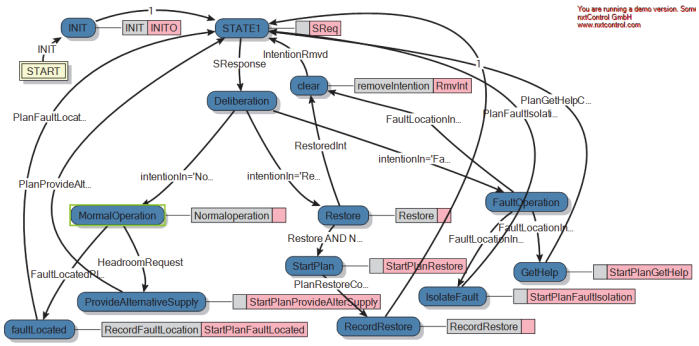

Fig. 6. Deliberation of the CSWI agent. ECC of the Interpreter FB.

D. Proposed hybrid architecture

In the proposed hybrid architecture there are two layers (machines): deliberative and reactive. Both machines are running concurrently as depicted in Fig. 8. Layers synchronize by setting or reading a synchronization variable - *flag*, a Boolean variable. Only reactive layer sets the flag, deliberative layer only reads it. When the reactive layer recognizes the situation to react, it signals to the deliberative layer by setting the flag. The deliberative layer suspends its execution and waits for the flag to be cleared. Once the reactive layer completes its set of actions, it clears the flag. The deliberative layer resumes its execution and reviews the goal at the top of the intention stack considering updated beliefs. The updated beliefs will reflect the changes in the environment and internal variables of the agent after the actions of reactive layer. In this way reactive module is given higher priority, since it can set/clear the *flag* as it needs.

Let R denote the reactive layer and D the deliberative layer. R and D are formally described as tuples:

$R = (Q_R, \Sigma_R, q_R^0, \rightarrow_R)$ and $D = (Q_D, \Sigma_D, q_D^0, \rightarrow_D)$, where

$Q_R = \{R1, R2\}$, $Q_D = \{D1, D2\}$ are finite sets of states of respective machines;

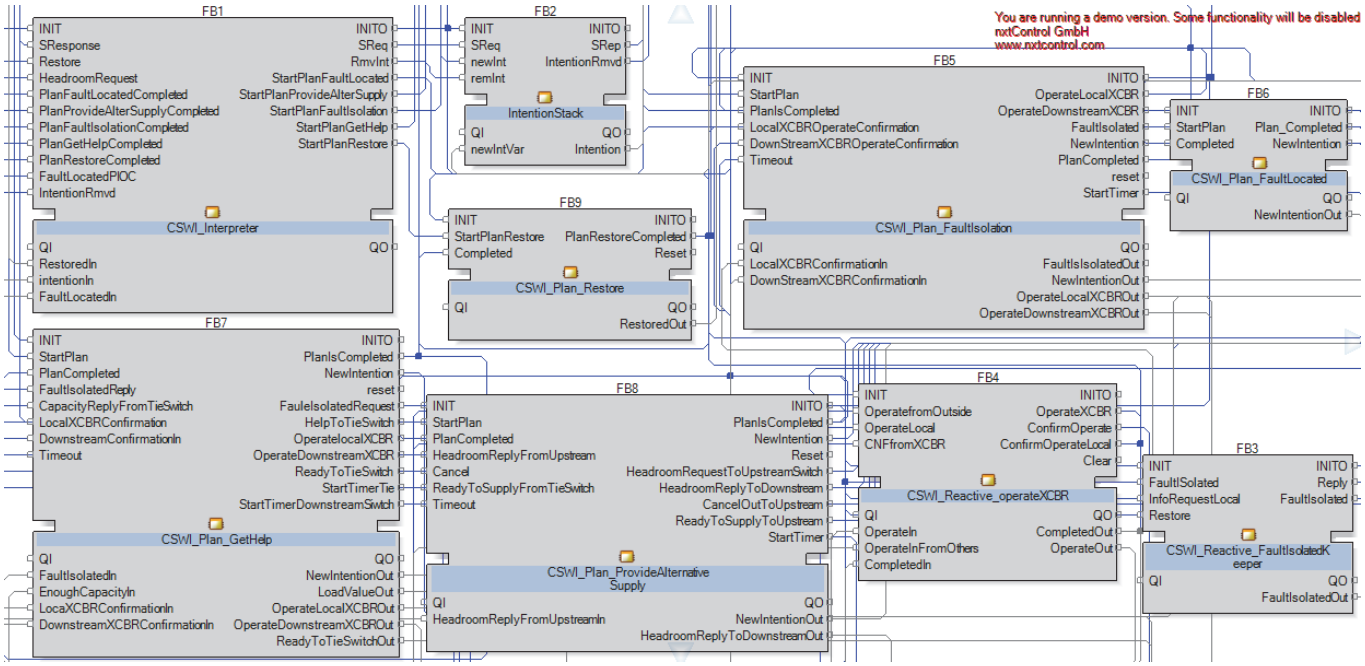

Fig. 7. CSWI agent. Internal structure: reactive behaviours, plans, intention stack and interpreter.

$q_R^0 = R1,\ q_D^0 = D1$ are initial states ;

$\Sigma_R = \{a, b\}, \Sigma_D = \{a, b\}$ are the finite alphabets of the machines, which are described later;

$\longrightarrow_R, \longrightarrow_D$ are transition functions, such that

$\longrightarrow_R : Q_R \times \Sigma_R \to Q_R$, and $\longrightarrow_D : Q_D \times \Sigma_D \to Q_D$,

mapping a state and a letter from the alphabet, to a state.

The *flag* is set, when it is assigned value "*true*", and *flag* is cleared, when it has value "*false*".

$flag = true$; - flag is set. This is denoted as *a*.

$flag = false$; - flag is cleared. This is denoted as *b*.

Then, *a* and *b* compose alphabets of the reactive and deliberative machines, i.e. $\Sigma_R = \Sigma_D = \{a, b\}$ is a finite alphabet of corresponding machines.

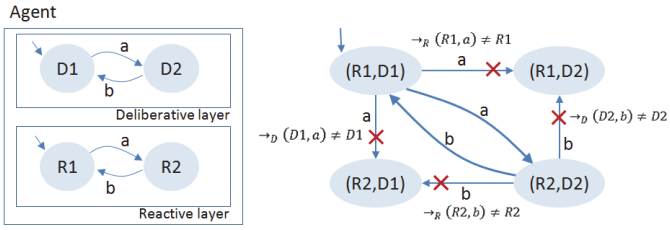

Fig. 8. System with concurrent processes R and D, and their parallel composition.

For a transition representation in the form $\to (q_1, a) = q_2$, the following short hand notation will be used $q_1 \overset{a}{\to} q_2$.

State R1 is the state when the reactive machine monitors the inputs. When the inputs require an action, reactive machine moves to the state R2, where it performs actions. The deliberative machine starts in state D1. This state is normal operation, while state D2 is entered when the reactive machine sets the *flag*. The deliberative layer returns to the state D1 when *flag* is cleared. Both machines have the same set of traces (action sequences) $L = \{ab\}^* \cup \{ab\}^* \times \{a\}$.

In the proposed agent, the layers exist in a parallel composition and so they construct the hybrid architecture as a synchronous product of the concurrent machines. Reactive and deliberative machines synchronise on transitions with common alphabet *{a, b}*.

The semantics of the resultant system is

$[\![Agent_{\|}]\!] = (Q_{\|}, \Sigma_{\|}, q_{\|}^0, \longrightarrow_{\|}) = R \parallel D$.

The transition function $\longrightarrow_{\|}$ has σ-transition if the following condition is satisfied.

For any state $(q_1, q_2) \in Q_{\|}$, for every $\sigma \in \Sigma_{\|}$ such that $q_1 \overset{\sigma}{\to} q_1'$ and $q_2 \overset{\sigma}{\to} q_2'$, then $(q_1, q_2) \overset{\sigma}{\to}_{\|} (q_1', q_2')$.

From all the possible transitions of the composition $Agent_{\|}$, only transitions between states *R1D1* to *R2D2* satisfy above definition. Thus,

$(R1, D1) \overset{a}{\to} (R2, D2)$, and $(R2, D2) \overset{b}{\to} (R1, D1)$.

Fig. 8 demonstrates the parallel composition of reactive and deliberative machines. The set of action sequences or traces of the hybrid architecture is $L_{\|} = \{ab\}^* \cup \{ab\}^* \times \{a\}$.

The section has presented the formal description of the proposed automation agent architecture, including deliberative and reactive layers and their combination. Combining these two architectures into a system may help satisfy requirements of resource limitation and time constraints of the automation agents, by utilizing pre-compiled library of plans and intentions, and modelling reactive behaviours with FSMs.

As an example of agents with hybrid architecture, Fig. 7 depicts a CSWI agent. The figure shows the internal structure of the CSWI agent. In this case deliberative and reactive layers do not perform competing actions, therefore there is no need for a synchronisation mechanism.

In order to demonstrate one of the implementations of synchronisation mechanism, let us assume CSWI agent has slightly different goals, and so reactive and deliberative layers perform competing actions. The corresponding FBs have been modified to illustrate the example.

Fig. 9 illustrates the synchronization mechanism of the CSWI agent.

As it can be seen, the deliberative layer does not set *flag* and does not output the *flag*. The deliberative layer only reads the *flag* and only acts when the *flag* is cleared. As seen from the ECC of *CSWI_Interpreter*, every deliberation state is taken only if *flag* is cleared. The machine suspends at the states *start* and *Deliberation*. The deliberative machine does not stay in the planning states, it signals a plan and moves back to the *start* state. A plan executes a single action every time it receives *DoPlan* event from the interpreter. Thus, a plan completes a single action and stops, waiting for the interpreter. The interpreter will not leave "*start*" state if *flag* is set. In this way the deliberative machine suspends its actions. If *flag* is set when the interpreter is in the *Deliberation* state, it will not leave the state until the *flag* has been cleared.

When the *flag* is cleared, the deliberation continues with the current intention. At the intention states, the machine deliberates with new updated beliefs. Here it can choose a different plan or evaluate the current intention to be impossible or succeeded and so get a new intention from the stack. If the *flag* is set, when *CSWI_Interpreter* is in intention states, then it will return to the *start* state and wait for flag to be cleared.

In the reactive layer, *behavior* modules will set the *flag* before they start their actions. Fig. 9 shows the *behavior OperateSwitch*. Before operating the switch, it sets the *flag*. On confirmation from the switch, the module completes its actions and returns to the initial state, where the *flag* will be cleared. *Flags* from each behavior are merged with an *OR* block. This ensures that *flag* is set as long as at least one of the behavior modules keeps it set.

The next section will describe the implementation of the proposed architecture.

IV. Automation Agent in a Distributed FLISR Application

The above agent architecture was applied on a FLISR application, which was developed with the earlier version of the architecture in [21]. The short description of the scenario is presented below; more details can be found in [22].

A. The FLISR scenario description

The sample distribution network is depicted on Fig. 10. The distribution utility consists of three 11kV feeders supplied by three different zone substations. Distribution substations are positioned along each feeder as demanded by customers' loads. In the initial state the switches ROS3, ROS4 and ROS 9 are open. All other switches are closed.

The scenario begins with a tree falling on the 11kV mains, causing a permanent fault on feeder F1 between CB1 and ROS1. The feeder protection trips circuit breaker CB1 at zone substation B. Sectionalizing switches ROS1 and ROS2 do not register the passage of fault current. After an attempted automatic reclosure, CB1 goes to lockout. Switches ROS1 and ROS2 are no longer energized, and they propagate a "call for help" towards a zone substation of the adjacent feeders. CB2 at zone substation A, and CB3 at zone substation C, respond with information about the headroom (excess capacity) available. This information propagates back down feeders F2 and F3. Switches ROS3 and ROS4, compare the available excess capacities with their respective loads. The switches agree on the steps necessary to restore supply: The mid-section of feeder F1 will transferred to feeder F2; the tail-section will be transferred to feeder F3; the head section will have to await repair. In the meantime, the control centre sends the crew to repair the located fault and then sends a command to the nodes of the system to restore pre-fault states.

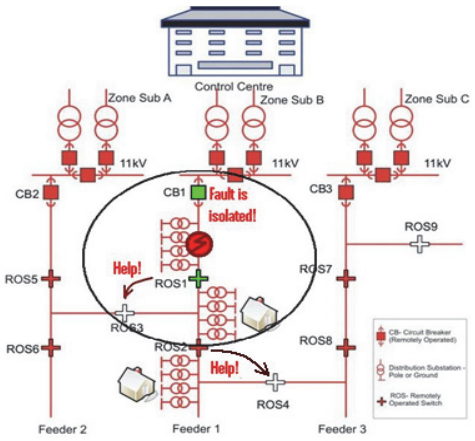

Fig. 10. Sample distribution network. Fault events and actions.

To simplify permutations of this scenario, only good paths are modelled (assume that all operations are successful).

B. Implementation of the agent architecture

The distributed components of the system collaborate to perform fault location, fault isolation and supply restoration tasks. The automation functions identified for the given scenario are overcurrent protection, protection trip conditioning, autoreclosing, switch control, circuit breaker, switch and current measurement. These functions are modelled with the following LNs: PIOC, PTRC, RREC, XCBR, CSWI, XSWI, TCTR and CILO. CILO LN was not considered in the implementation to simplify the scenario.

Modelling an agent-based system for the scenario is straight forward following the proposed architecture, where

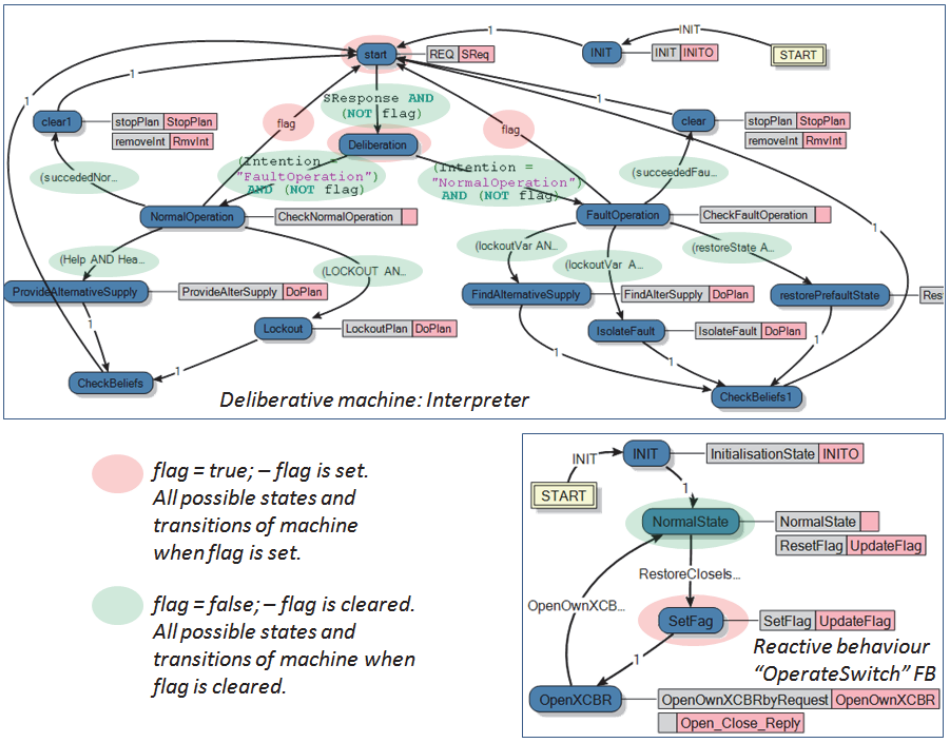

Fig. 9. CSWI agent. Synchronisation between deliberative and reactive layers. ECC of the CSWI_Interpreter FB and ECC of the Plan_OperateSwitch FB are presented.

each LN maps to an agent. Fig. 11 presents FB network for the CB1 section of developed agent-based system.

Agents TCTR, XCBR, XSWI, RREC, PTRC are reactive agents and have only the reactive layer. These agents are not involved in the collaborative tasks mentioned above. More interesting examples are agents CSWI and PIOC. They have both layers: reactive and deliberative. PIOC is responsible for fault detection and location tasks. CSWI agent performs fault isolation and supply restoration tasks.

An example of pure reactive agent, is RREC modelled with *RREC_Reactive* FB. This agent performs the auto-reclosing function of the circuit breaker. On a trigger from PTRC agent, that indicates detection of the fault and tripping of the circuit breaker, RREC starts the so-called "dead timer". This is the time is given for any temporary faults to be cleared before the RREC attempts to re-close XCBR. RREC closes the XCBR and starts the reclaim timer. If the detected fault is temporary, then the reclaim time elapses and RREC comes back to normal state. However, if the fault is permanent, then the XCBR will trip the second time, before the timer expires. In this case the RREC agent goes to lockout state and informs local PIOC agent about the permanent fault. When permanent fault has been cleared, on the command "*Restore*" RREC agent returns to the normal state. As seen from the example, the *behaviours* contain simple actions. Simple ECC are easier to understand, debug and verify.

Agents with hybrid architecture are CSWI and PIOC. CSWI agent is described above in section III.

The PIOC agent has the same architecture with its own library of plans and reactive behaviours.

The deliberative layer performs procedural reasoning, given a library of plans and intentions, while the reactive layer performs time critical operations and modelled as ECC (i.e. FSM) of the basic FBs.

Instantiating these LN agents following the system topology, allows modelling the FLISR application for the whole distribution system. The agents of the same type, e.g. CSWI are the instances of the same FB *CSWI_agent*. Fig. 11 presents excerpt of the system for position of CB1. The agents at each ROS and CB have the same set of FBs and interconnections as CB1. This can be seen at the system level scale of the complete agent-based system given in Fig. 12. This demonstrates re-use of software components (agents).

The developed agent-based system was tested using a co-simulation framework. The following section will present the simulation set up and the results of the experiment.

C. Discussion and results

The described use case observes that agents in power system automation may have only a reactive layer, a deliberative layer or both.

Another observation is that an LN, modelling an automation function, is highly specialised. Therefore, the respective agent inherits well-defined goals, plans and actions. This provides a domain engineer with well-defined framework to design an agent's model and its behaviour.

The proposed agent architecture is designed for the application domain. It allows the capture of power system automation concepts naturally, rather than forcing the problem domain to fit a generic agent architecture.

As shown in this case study, IEC 61499 is a migration pathway for agent-based solutions to industrial practice. The reported agent-based FLISR solution was developed in nxtStudio and first was simulated using IEC 61499 softPLCs and then deployed to industrial hardware ADAM 6650 of Advantech.

The testbed for the experiment is depicted on Fig. 12. The simulation set up consists of six devices and a PC for displaying HMI. The devices and a PC are networked with Ethernet. The agents that correspond to a feeder are deployed to one device, and agents that belong to a tie switch section are also deployed to one device. Therefore, there are five devices. The sixth device is softPLC, executing FBs related to HMI. The experiment was conducted using co-simulation framework [23]. The agent-based FLISR application is coupled with the Matlab model of the distribution system. Agents XCBR and TCTR are interfaced to the circuit breakers and current measurement units correspondingly. Communication is performed using UDP communication function blocks. Matlab simulation runs in parallel with the IEC 61499 agent-based system. More details on the co-simulation framework can be found [23].

The fault is simulated on the section of circuit breaker CB1 (Fig. 10), as described in section IV. The simulation results are shown on the Fig. 12. The fault was cleared in

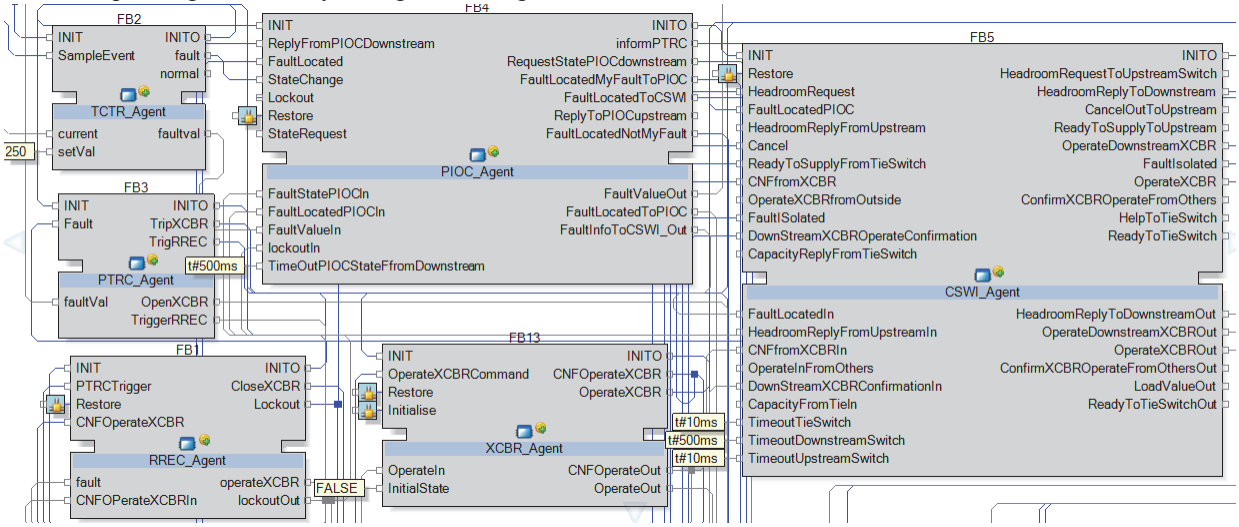

Fig. 11. Agent-based FLISR application. An agent network for section CB1 is shown only.

0.03 seconds of the simulation time after the reclosure. Supply to sections ROS1 and ROS2 was restored almost immediately, in 0.025 seconds of the simulation time. As it can be seen, the sections ROS1 and ROS 2 have restored the supply from feeders 2 and 3 correspondingly. It has to be noted, that the simulation time is of a different scale from the real time. The fault clearance time and time taken to restore supply in this experiment is dependent on the computational power of the processor to perform Matlab simulation and run softPLCs, and also on communication delays. What is important in this experiment is that the agents of the system were able to successfully perform FLISR algorithm in a distributed collaborative manner.

The experiments were conducted simulating fault on sections ROS1 and ROS2, and in both cases the fault was correctly located, isolated and healthy sections had the alternative supply restored correspondingly.

D. Comparison of the implementations

The other implementation of the FLISR with IEC 61499 is presented in [21, 22]. The work studied the feasibility of the industrial agent based on IEC 61499 and IEC 61850. The paper demonstrated that agents modelled as "intelligent LN" (iLN) are capable of performing FLISR actions without central control intervention.

An CSWI agent developed in [21] is depicted in Fig. 13. As it can be seen, the agent does not have a well structured architecture. It is a result of "ad-hoc" prototyping, customised for the problem. This "ad-hoc" approach is widely exploited in the works on agents for power system automation as reported in the literature.

The system in [21] was developed with no intention of further re-use or extension. The system cannot be re-produced, the agents are unique for this solution. As seen from the Fig. 13 behaviour of the CSWI agent is not explicit.

In contrast, the proposed implementation of the CSWI agent has well defined architecture. Fig. 7 shows clearly two reactive behaviours, five plans, the intention stack and the interpreter. The design process is methodological and therefore agents are re-producible. From Fig. 7 the behaviour of CSWI agent can be understood.

The complexity of the CSWI agents in both works have been measured using McCabe's metrics [24]. This estimates cyclomatic complexity of the ECC of each FBs within the CSWI agent. CSWI agent in [21] implementation achieved average score of 4. Similar results have been obtained for proposed CSWI agent - 3.8. Good practice is when McCabe's complexity measure is below 10. The proposed formal architecture, although resulted in a greater number of FBs within CSWI, did not affect overall complexity of the agent.

The benefit of the proposed architecture also can be realized when maintaining, extending or changing the agent behaviour.

All components of the proposed agent architecture are loosely coupled. Table 1 illustrates coupling within the proposed CSWI agent. The coupling is estimated as a number of input and output events and data connections between the corresponding FBs. As it can be seen the plans do not affect each other. Some plans push new intentions into the stack. Interpreter and plans often have 2 connections, to start the plan and inform of plan completion. Reactive behaviours often do not interact with the deliberative layer at all. In this case, plan *FautIsolation* sends operate command to reactive behaviour *OperateXCBR*, and receives the confirmation creating 4 connections including data for each event. Interpreter adds and deletes intentions from the intention stack, and requests for the top intention.

Thanks to the loose coupling of components, the proposed agent architecture is scalable and maintainable. For instance, introduction of a new plan will only affect the interpreter, adding an additional intention state and a transition.

In contrast, in [21], CSWI agent's decision making tightly couples various behaviors. A single FB combines several behaviors, e.g. *Fault* FB (Fig. 13). *Fault* performs behaviors of "fault isolation" and "search for alternative supply". Each additional behavior to the existing structure of CSWI agent, will require a unique approach, addressing challenges of its

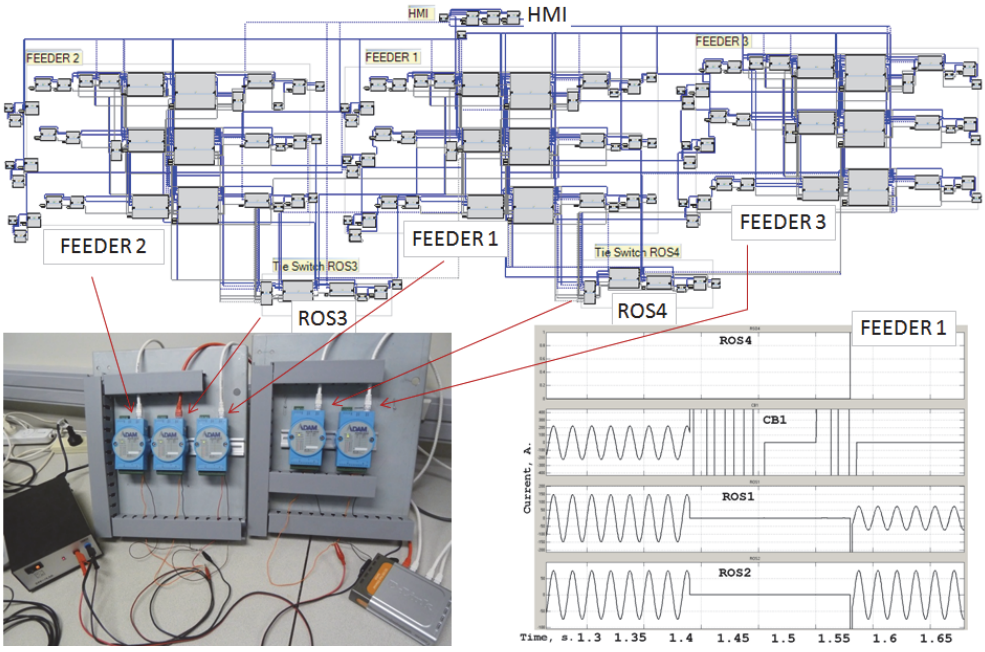

Fig. 12. The testbed architecture used for agent-based FLISR application.

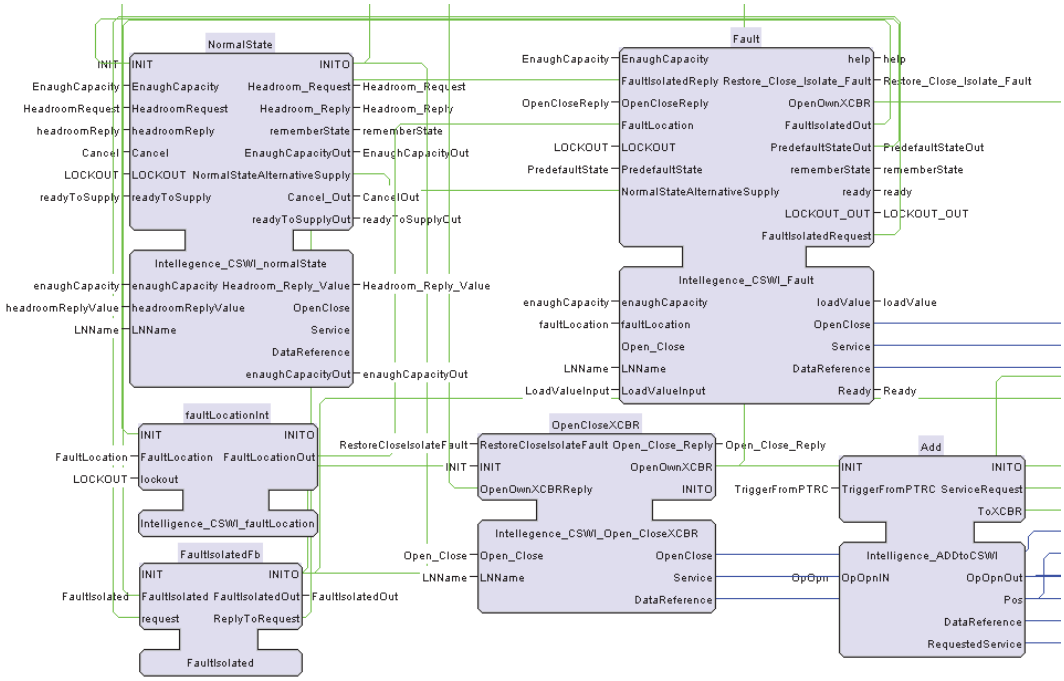

Fig. 13. CSWI agent in FBDK implementation.

integration with the existing components. An introduction of a new behavior will have an unpredictable effect on the agent.

V. Conclusion

This work is aimed at facilitating industrial adoption of the agent technology. The paper proposed an agent architecture based on industrial standards IEC 61850 and IEC 61499.

The designed agent-system is functionally complete: agents handle faults at any location within the distribution network. Functional requirements were captured with defined LNs, which are the base of the agent architecture. In this way, the architecture ensures that the requirements are carried through to the final system design.

The control system designed with the agent technology is scalable and flexible. The architecture simplified FLISR design, by reusing software components implementing agents. The agents are instantiated according to the LN types of primary devices that constitute each feeder and required automation functions. Such an approach can be taken to re-design the system for new topologies or changes in requirements.

References

[1] M. Wooldridge, *An introduction to Multiagent Systems*. UK: John Wiley & Sons Ltd, 2009.

[2] M. Wooldridge and N. R. Jennings, "Intelligent Agents: Theory and Practice", *The knowledge Engineering Review*, vol. 10, no. 2, 1995.

[3] P. Leitao, V. Marik, and P. Vrba.(2013). Past, Present, and Future of Industrial Agent Applications. *IEEE Trans. Ind. Inf.* [Online]. 9(4), pp. 2360-2372. Available: http://ieeexplore.ieee.org/xpl/articleDetails.jsp?tp=&arnumber=6319392

[4] S. Bussmann, N. R. Jennings, and M. Wooldridge, *Multiagent systems for manufacturing control: A design methodology*. Germany: Springer-Verlag, 2004.

[5] L. S. Sterling and K. Taveter, *Art of the agent oriented modeling*. USA: MIT, 2009.

[6] P. Vrba, P. Tichy, V. Marik, K. H. Hall, R. J. Staron, F. P. Maturana, and P. Kadera.(2011). Rockwell Automation's Holonic and Multiagent Control Systems Compendium. *IEEE Trans. Sys., Man, and Cybern., Part C: App. and Reviews.* [Online]. *41(1)*, pp. 14-30. Available: http://ieeexplore.ieee.org/xpl/articleDetails.jsp?tp=&arnumber=5545420

[7] P. Ferreira, S. Doltsinis, A. Anagnostopoulos, F. Pascoa, and N. Lohse. (2013). A performance evaluation of industrial agents.

TABLE 1. COUPLING BETWEEN CSWI AGENT COMPONENTS.

CSWI Agent components	Interpreter	Intention stack	Plan Get Help	Plan Provide alternative supply	Plan Fault Located	Plan Restore	Plan Fault Isolation	Reactive Operate XCBR	Reactive Fault Isolated Keeper
Interpreter	n/a	-	-	-	-	-	-	-	-
Intention stack	5	n/a	-	-	-	-	-	-	-
Plan Get Help	2	2	n/a	-	-	-	-	-	-
Plan Provide alternative supply	2	2	**0**	n/a	-	-	-	-	-
Plan Fault Located	2	2	**0**	**0**	n/a	-	-	-	-
Plan Restore	3	0	**0**	**0**	**0**	n/a	-	-	-
Plan Fault Isolation	2	2	**0**	**0**	**0**	**0**	n/a	-	-
Reactive Operate XCBR	0	0	4	0	0	0	4	n/a	-
Reactive Fault Isolated Keeper	0	0	3	0	0	0	1	0	n/a

Presented at IECON. [Online]. Available: http://ieeexplore.ieee.org/xpl/articleDetails.jsp?tp=&arnumber=6700365

[8] S. Ulewicz, D. Schutz, and B. Vogel-Heuser. (2012). Design, implementation and evaluation of a hybrid approach for software agents in automation. Presented at ETFA. [Online]. Available: http://ieeexplore.ieee.org/xpl/articleDetails.jsp?tp=&arnumber=6489766

[9] I. Hegny, O. Hummer, A. Zoitl, G. Koppensteiner, and M. Merdan. (2008). Integrating software agents and IEC 61499 realtime control for reconfigurable distributed manufacturing systems. Presented at SIES.[Online]. Available: http://ieeexplore.ieee.org/xpl/articleDetails.jsp?tp=&arnumber=4577710

[10] W. Lepuschitz, M. Vallee, M. Merdan, P. Vrba, and J. Resch. (2009). Integration of a heterogeneous Low Level Control in a multi-agent system for the manufacturing domain. Presented at ETFA.[Online]. Available: http://ieeexplore.ieee.org/xpl/articleDetails.jsp?tp=&arnumber=5347061

[11] International Electrotechnical Comission, "IEC 61850 Communication networks and systems for power utility automation," ed. 2, Switzerland, 2009.

[12] *Fulnction blocks - Architecture*, IEC standard 61499-1, 2005.

[13] V. Dubinin and V. Vyatkin. (2006). Towards a Formal Semantic Model of IEC 61499 Function Blocks. Presented at IEEE Conf. on INDIN. [Online]. Available: http://ieeexplore.ieee.org/xpl/articleDetails.jsp?tp=&arnumber=4053354

[14] V. Vyatkin and H.-M. Hanisch (2001). Formal modeling and verification in the software engineering framework of IEC 61499: a way to self-verifying systems. Presented at the IEEE ETFA. [Online]. Available: http://ieeexplore.ieee.org/xpl/articleDetails.jsp?tp=&arnumber=997677

[15] G. Cengic and K. Akesson. (2010). On Formal Analysis of IEC 61499 Applications, Part A: Modeling. *IEEE Trans. Ind. Inf.* [Online]. *6(2)*, pp. 136-144. Available: http://ieeexplore.ieee.org/xpl/articleDetails.jsp?tp=&arnumber=5405074

[16] A. Luder and C. Schwab. (2005). Formal models for the verification of IEC 61499 function block based control applications. Presented at the IEEE ETFA. [Online]. Available: http://ieeexplore.ieee.org/xpl/articleDetails.jsp?tp=&arnumber=1612508

[17] Y. Li Hsien, G. D. Shaw, P. S. Roop, and Z. Salcic. (2012). Synthesizing Globally Asynchronous Locally Synchronous Systems With IEC 61499. *IEEE Trans. Sys., Man, and Cyber., Part C: App.and Reviews.* [Online]. *42(6)*, pp. 1465-1477. Available: http://ieeexplore.ieee.org/xpl/articleDetails.jsp?tp=&arnumber=6198367

[18] V. Dubinin and V. Vyatkin. (2007). On definition of a Formal Model for IEC 61499 Function Blocks. *EURASIP Journal on Ebmedded Systems.* [Online]. Available: http://jes.eurasipjournals.com/content/2008/1/426713

[19] M. Wooldridge, *Reasoning about Rational Agents.* MA: The MIT Press.

[20] G. Zhabelova, "Software architecture and design methodology for distributed agent-based automation of Smart Grid," Ph.D dissertation, Dept. of Elect. and Comp. Eng., Auckland Univ., Auckland, New Zealand, 2013.

[21] G. Zhabelova and V. Vyatkin. (2012). Multi-agent Smart Grid Automation Architecture based on IEC 61850/61499 Intelligent Logical Nodes. *IEEE Trans. Ind. Elect.* [Online]. *59(50)*, pp. 2351-2362. Available: http://ieeexplore.ieee.org/xpl/articleDetails.jsp?tp=&arnumber=6018303

[22] G. Zhabelova, "Investigation of intelligent smart grid automation implementation by means of IEC 61499 and IEC 61850," M.E. thesis, dept. Elect. and Comp. Eng., Auckland Univ., Auckland, New Zealand, 2009.

[23] C. Yang, G. Zhabelova, C. Yang, and V. Vyatkin. (2013). Cosimulation Environment for Event-Driven Distributed Controls of Smart Grid. *IEEE Trans. Ind. Inf.* [Online]. *9(3)*, pp. 1423-1435. Available: http://ieeexplore.ieee.org/xpl/articleDetails.jsp?tp=&arnumber=6494291

[24] T. J. McCabe. (1976). A Complexity Measure. *IEEE Trans. Software Eng*. [Online]. *SE-2(4)*, pp. 308-320. Available: http://ieeexplore.ieee.org/xpl/articleDetails.jsp?tp=&arnumber=1702388

Gulnara Zhabelova graduated with Diploma degree in Robotics and Mechatronics in 2006, received ME in Automation and Control degree in 2008 from Karaganda State Technical University, Karaganda, Kazakhstan and ME in Computer Systems in 2009 and PhD in Electrical and Electronics Engineering in 2013 from University of Auckland, Auckland, New Zealand. Currently she is with the Lulea University of Technology. Prior to this, she was with the Industrial Informatics Lab at The University of Auckland. Her current work is on application of ICT and agent technology in Smart Grid. Her interests are in Agent technology, its formal definition, theory and application in wide practical domain: Automation and Control, Protection in Energy generation, transmission, distribution and consumption; Building automation, Demand Side management, Advanced Metering Infrastructure (AMI), Energy markets and policies.

Valeriy Vyatkin (M'03–SM'04) received Ph.D. degree from the State University of Radio Engineering, Taganrog, Russia, in 1992. He is on joint appointment as Chaired Professor of Dependable Computation and Communication Systems, Luleå University of Technology, Luleå, Sweden, and Professor of Information and Computer Engineering in Automation at Aalto University, Helsinki, Finland. Previously, he was a Visiting Scholar at Cambridge University, U.K., and had permanent academic appointments with the University of Auckland, Auckland, New Zealand; Martin Luther University of Halle-Wittenberg, Halle, Germany, as we'll as in Japan and Russia.His research interests include dependable distributed automation and industrial informatics; software engineering for industrial automation systems; and distributed architectures and multi-agent systems applied in various industry sectors, including smart grid, material handling, building management systems, and reconfigurable manufacturing.
Dr. Vyatkin was awarded the Andrew P. Sage Award for the best IEEE Transactions paper in 2012.

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 2003, 2006, and 2010, he was awarded DAAD-grants to work as a Guest Scientist at Martin-Luther-University, Halle-Wittenberg, Germany. He held Visiting Researcher position at the University of Auckland, New Zealand (2011), and at the Lulea University of Technology, Sweden (2013, 2014). His research interests include formal methods for specification, verification, synthesis, and implementation of distributed and discrete event systems.