Smartlog - A declarative language for distributed programming in smart grids

Thi-Thanh-Quynh Nguyen, Vincent Debusschere, Christophe Bobineau, Quang Huy Giap, Nouredine Hadjsaid

Smartlog – A declarative language for distributed programming in smart grids

Thi-Thanh-Quynh Nguyen[a], Vincent Debusschere[a,*], Christophe Bobineau[b], Quang Huy Giap[c], Nouredine Hadjsaid[a]

[a] *Univ. Grenoble Alpes, CNRS, Grenoble INP[1], G2Elab, 38000 Grenoble, France*
[b] *Univ. Grenoble Alpes, CNRS, Grenoble INP[1], LIG, 38000 Grenoble, France*
[c] *Department of Electrical Engineering, University of Science and Technology, University of Danang*

Abstract

In the control and supervision of smart grids, the objective is to handle any change in the system as fast as possible, with as few resources as possible. In this context, this paper proposes a new language, called *Smartlog*, designed with a declarative approach. This avoids collecting and analyzing data presenting no interest, and thus being less efficient in bandwidth usage and computational time. *Smartlog* is designed for operating smart grids defined as abstract structures of large and scalable distributed databases. From its definition, some major properties of this language, such as simplicity, incremental capacity, and scalability are highlighted. The language is tested on the application of the secondary control of an isolated microgrid using a real-time simulator connected to Raspberry Pis. The characteristics of *Smartlog* are illustrated thanks to a comparison with an imperative programming implementation of the same regulation.

Keywords: Smartlog, smart grids, declarative programming, distributed database system, secondary control, discrete-time control

1. Context

The power-sharing of renewable energy resources and storage devices increases regularly in the traditional power grid. This trend emphasizes the problem of monitoring and control, as the power system should be operated in the most reliable, efficient, and flexible way with increasing variability of both energy producers and consumers. Actually, due to the development of smart devices such as smart meters, micro-computers and the use of available communication infrastructures (e.g., WiFi [1], 3G, 4G, 5G, power-line communication [2]), the power system could be more closely controlled and monitored, even up to real-time. The complex interconnection between information, communication technologies and energy systems represent, even indirectly, a significant part of the definition of smart grids [3].

In that context, metadata (i.e., data providing information about other data) are being gathered with a rapidly growing rate. Data can come from energy sources, geographic positions, customer information, outages management,

*Corresponding author
Email addresses: thi-thanh-quynh.nguyen@g2elab.grenoble-inp.fr (Thi-Thanh-Quynh Nguyen), vincent.debusschere@grenoble-inp.fr (Vincent Debusschere), christophe.bobineau@grenoble-inp.fr (Christophe Bobineau)
[1] Institute of Engineering Univ. Grenoble Alpes

demand-response control systems [4], metering infrastructures [5], hierarchical stability control [6], as well as monitoring processes, and so on [7]. The quantity of collected data leads to a problem of volume and variety, which is at the core of the *big data* trend.

1.1. Smart grids, data production and handling

The shorter the data sampling time is, the more accurate and precise control decisions can be. But the enormous quantity of data received by servers in a short period of time can cause a problem of velocity in data processing, particularly when considering heterogeneous and complex systems like smart grids. It is expensive to maintain powerful servers, without talking about enhancing the bandwidth, while data is growing in size, density and variety.

Besides, data processing is critical. It can take a lot of time to transform data into information and help to make decisions. When considering a large power system at the scale of a country in Europe, a central server cannot make use of all data simply due to the nodal limitation in computing power.

To deal with that, current researches focus on applying the concept of distributed algorithms to reduce the volume of data collected by servers and make better use of the available local computing units [8, 9]. This presents a very promising perspective for smart grids in the upcoming years [10].

1.2. Existing solutions and limitations

The control and monitoring of smart grids in real-time require handling any variations in the power system immediately. Classic algorithms, in imperative programming, describe steps of solution ("how to do") rather than the reaction of the control system ("what to do if"). This can sometimes lead to a loss of information, or the collection and analysis of a vast quantity of data, where the same action would have needed significantly less computing power with a more efficient programming language.

To circumvent that problem, this paper proposes a new language, called *Smartlog*, based on declarative programming. *Smartlog* supports, by definition, distributed computing in real-time and database management and is being developed exactly to propose this features, dedicated to smart grids.

There exist several languages based on declarative programming. On the one hand, for example, Datalog is used as a logic language [11]. It expresses the relation between data in centralized databases. However, the communication between databases in the heterogeneous system is not properly described. NDlog [12] and Netlog [13], on the other hand, support communication between local databases, particularly by defining a network protocol, rather than the distributed computing mechanism. But, for both NDlog and Netlog, data are stored without identifiers. The structure of command is then still not explicit.

Obviously, data sent to a centralized server could probably be pre-processed at the level of the local unit. If the smart grid is organized as a distributed database system. Each local database can compute and analyze, by itself, part of the information to contribute to the main solution. It is a great idea to reduce communications and local computations to and for centralized servers. Several articles mention distributed databases for smart grids, but usually, they are

not making use of the local computing units [14]. Moreover, the abstract structures of local computing units are not thoroughly illustrated yet, except partially in the standard IEC-61 850 [15], which only defines the data structure of the substations and their communication protocol. Besides, most of the imperative languages do not support distributed programming for the distributed data system. To be more efficient, the distributed deployment in imperative language must rely on distributed algorithms.

In addition, it is to be noted that most of the calculations and communications in distributed programming are carried out sequentially (thanks to the time step definition) rather than following the actual meaning of data [5]. The addressed problem is not perspicuous because the algorithm states the order in which operations occur ("how to do") and not how to react to information ("what to do").

Nevertheless, elements in smart grids are almost all eventual. This paradigm leads to computations and communications redundancies. In that case, an algorithm able to react to a concrete problem is more efficient than some sequences that are created to manage pre-defined states of a system. Thus, in this context, algorithms supporting declarative paradigms are more convenient than those supporting imperative ones.

1.3. Problem statement

This paper considers a system of smart devices embedded in a smart grid organized as a distributed database. In this context, a new language is proposed, considering high-level programming, called *Smartlog*. *Smartlog* is a declarative language which supports distributed programming, developed directly for a distributed database system.

This article expands the descriptions of the language, started in [16]. In particular, the termination of the program is analyzed, focusing on the mechanism of support for distributed programming as well as on the definition of its convergence which is presented in Section 2. Section 3 proposes a distributed secondary control in an isolated microgrid as a case study of the *Smartlog* implementation. A general conception for programming in *Smartlog* is also provided in section 4. The specific quantitative evaluation of the *Smartlog* programming language is presented in Section 6 through results of the case-study which were obtained from the implementation in a real-time simulation platform presented in Section 5. Finally, Section 7 concludes the paper and discusses future works.

2. Smartlog language

Smart grids involve many heterogeneous smart devices in order to ensure a reliable, flexible, and self-healing operation of the electrical grid by updating and processing data regularly or even in real-time. We propose to abstract the whole computing nodes in smart grids as a distributed database system in which *Smartlog* is in charge of the manipulation of distributed data. The architecture of each node in such an ad-hoc network is presented below.

2.1. The architecture of a node in smart grids

Each smart device oversees the computation and communication for its node in the smart grid. For reference, Fig. 1 presents that common architecture.

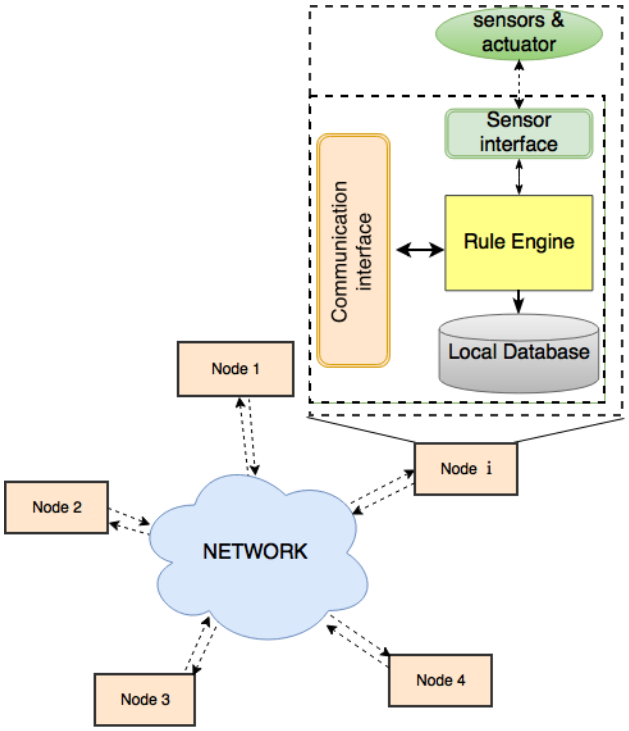


Figure 1: Architecture of each node within the considered structure of smart grid.

There are four main components in this architecture:

The local database: Stores the node's information such as parameters in the information infrastructure of the smart grid, set-point values (outputs), control parameters, and measured data which are collected from local sensors or directly from the grid. This mechanism limits data concentration on a central server as well as accelerates the time of access to data. Ultimately, this structure creates a smart grid as a system of distributed databases which is easily scalable [14].

The sensor interface: Is set up to collect data from sensors, storing everything into a local database and transferring the output values to the regulator in order to control any available active electrical components.

The rule engine: Is the most important part of the considered structure. That engine decides which rules are executed, relying on the programs written using *Smartlog*.

The communication interface: Is in charge of the interaction with other nodes over the communication network.

2.2. Structure of the language

The structure of *Smartlog* is described in three main parts, as expressed in Listing 1.

Listing 1: The structure of a Smartlog program.

```
Program(NameOfProgram){
Data_type{
//define the data types
}
Initial_data{
//set up initial data
}
Module(data_type 1){
//rules
}
Module(data_type 2){
//rules
}
...
}
```

2.2.1. *The "Data_type" block*

The data is declared in the form of a structure called the `data_type` that defines the schematic of the data stored in the database. It contains the name of the `data_type`, its attributes and the data type of the attribute. A *key* word is added after the data type of the attribute to indicate the critical attribute in the `data_type`. For example, the instruction of Listing 2 is used to declare a `data_type` called `Neighbor`.

Listing 2: Declaration of a "data_type" called "Neighbor".

```
Data_type{
Neighbor(NodeID:int key, NeighborID:int key, NeighborAddr:string).
}
```

The declaration of the `data_type Neighbor` in Listing 2 presents four attributes, two of which are critical: `NodeID` and `NeighborID`. They are assigned a *key* word after their data type.

The data used for distributed control in smart grids are divided into four types: the measured data, the parameters, the intermediate data, and the output. The measure data are collected by sensors. The parameters can be additional information in the communication network or constants used in the control process. The intermediate data holds the auxiliary variables used in the computation process, and the output data holds the results of computation and represents control variables.

2.2.2. *The "Initial_data" block*

Usually, initial parameters and output data are set up in this block. The stored data will not activate any rule in the `Module` blocks and thus are not mandatory in *Smartlog*. For example, the initial data for the `data_type` `Neighbor` is proposed in Listing 3.

Listing 3: Declaration of the initial data of the "data_type" called "Neighbor".

```
Initial_data{
Neighbor(1,2,'192.168.1.102').
}
```

2.2.3. *The "Module" block*

Rules. The definition of any command to be executed in each node is declared via rules. Each rule in *Smartlog* is defined in the form: "`head :- body`".

The `body` part, B, can comprise one or more terms $B = \{B_1, B_2, ..., B_n\}$. The terms are separated by a comma and can be a relational atom (`data_type`), conditional expressions, and/or assignments with arithmetic expressions (using the prefix "`:=`"). The first term in the body part and the head part, H, must be atoms.

When a rule is activated, the terms are checked one by one from left to right in the body part. If all terms in the `body` are validated, the execution of H is launched.

Variables. It is not necessary to declare variables in *Smartlog*, even if used in rules. The data type of a variable is defined automatically by considering its position in the `data_type`. The symbol "_" is used to indicate that one attribute is ignored in the `data_type`. Variables are used to store values of attributes. They are divided into two types: variables with assigned values (*linked variables*) and variables with values not yet assigned (*unlinked variables*).

All the attributes in the first term B_1 of the `body` are *linked variables*. The *unlinked variables* are defined relying on at least one *linked variable* in the atom. If the *linked variable* corresponds to the whole set of *key* attributes, there is only one possible value for *unlinked variables*. Otherwise, all possible values must be checked for *unlinked variables*. The `head` part needs then to be executed with each value assigned to variables. Listing 4 illustrates the definition of variables in *Smartlog*.

Listing 4: Exemple of variables definition in Smartlog.

```
Measure(j,v,i) :- Measure(id,v,i), Neighbor(id,_,j,_).
```

In Listing 4, all variables of the first term id, v, i are *linked variables*. The second term is the relational atom `Neighbor`, which has two key attributes: `NodeID` and `NeighborID`. The variable `j` holds the value of the `NeighborID` attribute, which is an *unlinked variable*. The execution of the `head` will generate a quantity that depends on the number of neighbors of the considered node.

Operators. The *negation* operator is denoted "~" in *Smartlog*. It is used to check whether it exists or not a value of the *linked variables* in the atom, as illustrated in Listing 5.

Listing 5: The "negation" operator in Smartlog.

```
Measure(id,v,i) :- Measure(id,v,i), ~Neighbor(id,_,_).
```

The *consumption* operator is denoted "!" in *Smartlog*. It is used to delete the atom of the *linked variables*, as illustrated in Listing 6. This operator allows managing a possible overload of the local data storage capacities.

Listing 6: The "consumption" operator in Smartlog.

```
Measure(j,v,i) :- Measure(id,v,i), !Neighbor(id,j,_).
```

The body part is terminated by a dot or a semi-colon. If it is a dot, the program will exit the current module after the execution of the rule. Otherwise, the next rules in the current module are performed.

Communication and storage. The `head` part contains the variables with assigned values in the `body` part and defines the execution of the rule(s). The execution can be by default an insertion or an update of the data. Sending data to another address is denoted by the symbol "^". Two actions executed simultaneously can be triggered using the symbol "&". To establish a communication, the shipping address should be marked with the symbol "@" in front of the *address variable* noted `j` in the example of Listing 7.

Listing 7: Communication and storage in Smartlog.

```
^Measure(id,v,i) :- Measure(id,v,i), Neighbor(id,_,@j).
```

To optimize the execution of the language, rules with the same first `data_type` placed in the `body` are grouped into a `Module`. A *Smartlog*-based program can then present many `Modules`, each consisting of many rules.

The name of the module is the same name as the `data_type` of the first term B_1. If there is any change in this `data_type`, the inside rules are performed sequentially. The changes in data are eventual. Modules are executed in parallel, which makes the order of programs undefined. The *Smartlog* language supports declarative programming to that purpose.

2.3. Ending a one-round execution of a Smartlog program

A set of variables for the rule R (with its body part B and its head part H) is called $Var(R)$. These variables are categorized in two types: uninterpreted and arithmetic, which are symbolized by $(\mathbb{N}, \leq)$ and $(\mathbb{R}, +, \times, \leq)$ respectively [13]. Considering a set of local data called I, a mapping on I from all variables of a rule R, $(Var(R))$ to $\mathbb{N} \cup \mathbb{R}$ is noted Θ_I. Mappings on I from a set of the variables in the body part $Var(B)$ and the head part $Var(H)$ are named φ_I and τ_I respectively, defined as:

$$\varphi_I(B) = \{\varphi_I | \varphi_I \subset \Theta_I, \forall B_i, \varphi_I(Var(B_i)) = B_i\} \tag{1}$$

$$\tau_I(H) = \{\tau_I | \tau_I \subset \Theta_I, \forall x \in Var(H), \varphi_I(x) = \tau_I(x), \varphi_I = B\} \tag{2}$$

$$B = \bigcup_{i=1}^{n} B_i \tag{3}$$

The execution of the "consumption" operator of a relational atom $!D(d_1, d_2, ..., d_n)$ deletes a portion of data instances, and modifies the set of local data I. So, after this execution, I is assigned as expressed in (4).

$$I := \{I \setminus \Delta_R^-(I), \Delta_R^-(I) = D(\varphi_I(d_1), \varphi_I(d_2), ..., \varphi_I(d_n))\} \tag{4}$$

J is a set of incoming data via the interfaces. The result of the rule R, produced in the set of data $(I \cup J)$ is $\Delta_R^+(I \cup J)$, with:

$$\Delta_R^+(I \cup J) = \tau_{I \cup J}(H) \tag{5}$$

There are two main executions for the head part. In the case of the default execution, the storage operator of the rule R is denoted $\Phi_R^{\downarrow}(I, J)$ and is expressed in (6).

$$\Phi_R^{\downarrow}(I, J) = I \cup J \cup \Delta_R^+(I \cup J) \setminus \Delta_R^-(I \cup J) \tag{6}$$

In the case of the communication execution, the sending operator of the rule R is noted $\Phi_R^{\uparrow}(I \cup J)$ and defined as:

$$\Phi_R^{\uparrow}(I \cup J) = \Delta_R^+(I \cup J) \tag{7}$$

A *Smartlog* program P can have many rules. With a set of incoming data J, the rules in P are evaluated and some are executed.

One-round execution. In a computing node α, on a local data set I and an incoming data set J, the one-round execution of P is given by a sequence of $(I_i^\alpha, \mathcal{P}_i^\alpha)_{i=0,1,...}$. With I_i^α and $\mathcal{P}_i^\alpha$ are the local data set and data to send of node α at step i, respectively. They are defined as follows:

$$\begin{cases} I_0^\alpha = \Phi_P^{\downarrow}(I, J) \\ I_{i+1}^\alpha = \Phi_P^{\downarrow}(I_i^\alpha, \emptyset) \text{ for } i \geq 0 \end{cases} \quad \text{and} \quad \begin{cases} \mathcal{P}_0^\alpha = \Phi_P^{\uparrow}(I \cup J) \\ \mathcal{P}_{i+1}^\alpha = \Phi_P^{\uparrow}(I_i^\alpha) \cup \mathcal{P}_i^\alpha \text{ for } i \geq 0 \end{cases} \tag{8}$$

One-round computation. In a computing node α, on $I \cup J$, a one-round computation (containing many one-round executions) of P terminates if all its non-deterministic one-round executions converge to a fixpoint, it means that $(I_i^\alpha, \mathcal{P}_i^\alpha) \overset{i\to\infty}{\to} (I^\alpha, \mathcal{P}^\alpha)$. Such a limit is called a *one-round fixpoint* of the program P in the node α.

When a local one-round computation l starts, the node α has a local instance $I^\alpha(l)$, the data receiving $J^\alpha(l)$. The new local data instance after that l^{th} one-round computation is $I^\alpha(l+1) = \lim_{i\to\infty} I_i^\alpha$ and set of sending data $\mathcal{P}^\alpha(l+1) = (\lim_{i\to\infty} \mathcal{P}_i^\alpha)$. So, in a local database, the termination of the *Smartlog* program relies on the convergence of the sequence of fixpoints.

2.4. How Smartlog supports distributed programming

Rules in a *Smartlog* program are grouped to define all actions of the system as a modification of a specific `data_type`. The measured data are collected and stored in a local database. Intermediate data, created during rules executions, can be shared. The data transferred between nodes are in the form of a `data_type`. The sending data allow triggering the next calculations in another node. As an illustration, Listing 8 and Listing 9 propose two modules in two nodes.

Listing 8: Module "A" in node *i*.

```
Module(A){
^TmpC(i,v,c) :- A(i,v,c), B(i,@j); }
```

Listing 9: Module "TmpC" in node *j*.

```
Module(TmpC){
C(i,v,c) :- TmpC(i,v,c), c<100; }
```

When the rule in Listing 8 of the node *i* is executed, the atom named `TmpC` is sent to node *j*. The module of Listing 9 in node *j* is performed after receiving `TmpC`. This mechanism is the core of the full support of *Smartlog* for distributed programming. The convergence of a distributed *Smartlog* program is thus defined as follows [13]:

Definition 2.1. Given P a centralized *Smartlog* program, $\mathcal{V}_\mathcal{G}$ a set of computing nodes in the distributed system and $I^\alpha, (I^\alpha \subset I)$ a data instance distributed in each node $\alpha \in \mathcal{V}_\mathcal{G}$, all one-round computations l of P converge to a fixpoint, i.e. all sequences $(I_i^\alpha(l), \mathcal{P}_i^\alpha(l)) \overset{i\to\infty}{\to} (I^\alpha(l), \mathcal{P}^\alpha(l))$ and all sequences $(I^\alpha(l), P^\alpha(l))$ have a limit $(I^\alpha, \mathcal{P}^\alpha)_{l\to\infty}$. This collection of limits $(I^\alpha, \mathcal{P}^\alpha)$ represents the *distributed fixpoints* of the program P.

3. Case study: Secondary control for isolated microgrids

3.1. Motivation

A *Smartlog* implementation should follow the node architecture of the distributed database system in the considered smart grid, handle the mechanism of the distributed programming and process the quantitative evaluations of the *Smartlog* language. Those points are validated with the help of the deployment of a "manually distributed" smart grid application in a real-time platform.

Measured data are stored near collection devices, and a declarative approach is used for expressing distributed data manipulations. The computations and communications are triggered for any change in the measure data (violation of the set-point value of the frequency for instance). Real-time applications, which require very short response times and small communication delay, are not the context of this paper. Herein, we assume to develop smart grid scenarios with the available communication infrastructure (i.e., 3G, 4G, wireless) having known delay time and a network of computing units.

3.2. The application support

As the objective is to illustrate the principle of the *Smartlog* language, the simulation model is reduced to a basic grid, composed of three distributed energy resources (DER) connected to AC static loads, as presented in Fig. 2. The PV systems and storage devices are considered as DC sources in which the generated powers are perfectly controllable.

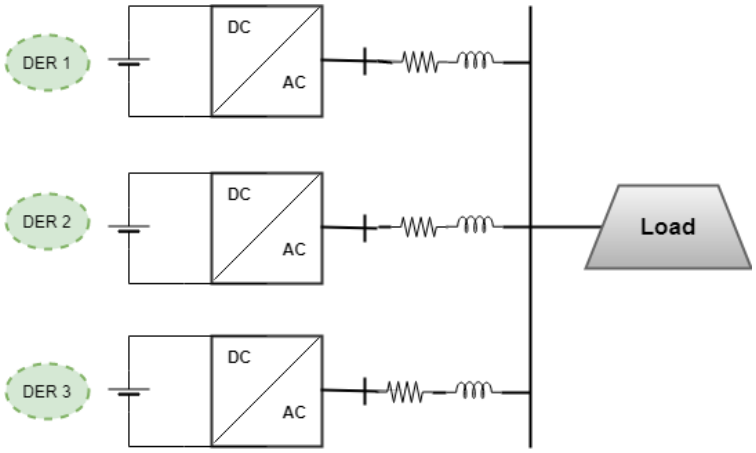


Figure 2: Isolated microgrid as a support to the illustration of *Smartlog* language.

In isolated mode, the microgrid is disconnected from the main grid (thus no PCC has to be modeled). All generators must still cooperate to maintain the power balance. The model of each voltage source inverter (VSI) supports a local primary control to properly operate the microgrid [17]. Each VSI includes three controllers: power, current, and voltage. The droop control is applied to the power controller, in order to represent the relationship between the electric parameters (P/f and Q/V). It is then emulated by transferring the functions in a Simulink/Matlab environment for the experimentation. The details of the converter design, as well as the parameters such as for the low-pass filters, droop characteristics and PI controllers, are referenced in [18].

3.3. The secondary control

When the local active power (P) or the reactive power (Q) of the load varies, the primary control of the generators reacts immediately to adjust its set-points so that the system keeps operating with an acceptable power balance and remains stable [19]. This leads to variations in frequency and voltage magnitudes, which impact the quality of the electricity supplied to the load. That is particularly true in isolated grids.

The main purpose of the secondary control is to restore the frequency and voltage magnitudes to their original set-point values after deviation, for instance, due to the primary control. The diagram of the secondary control in the VSI is shown in Figure 3.

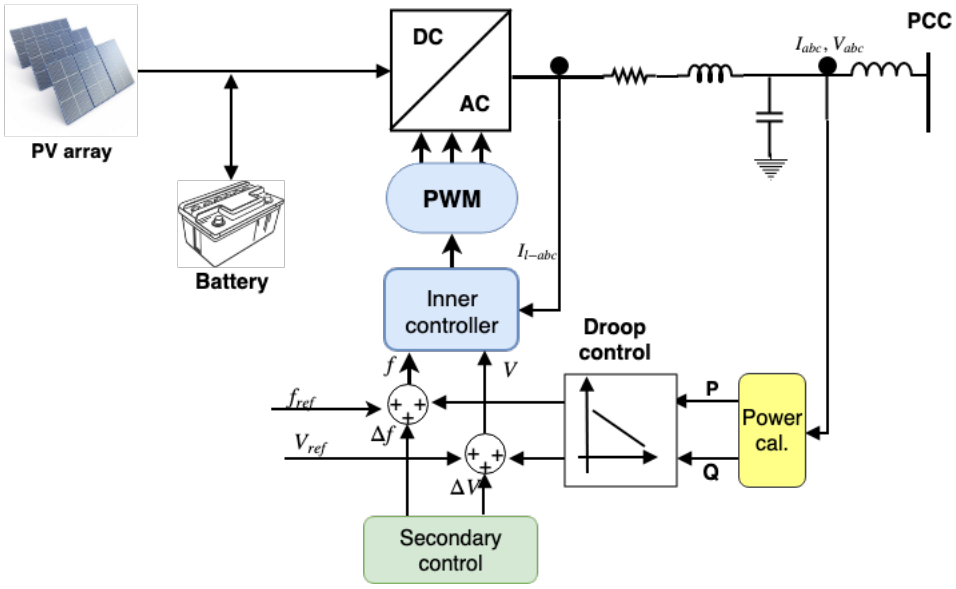
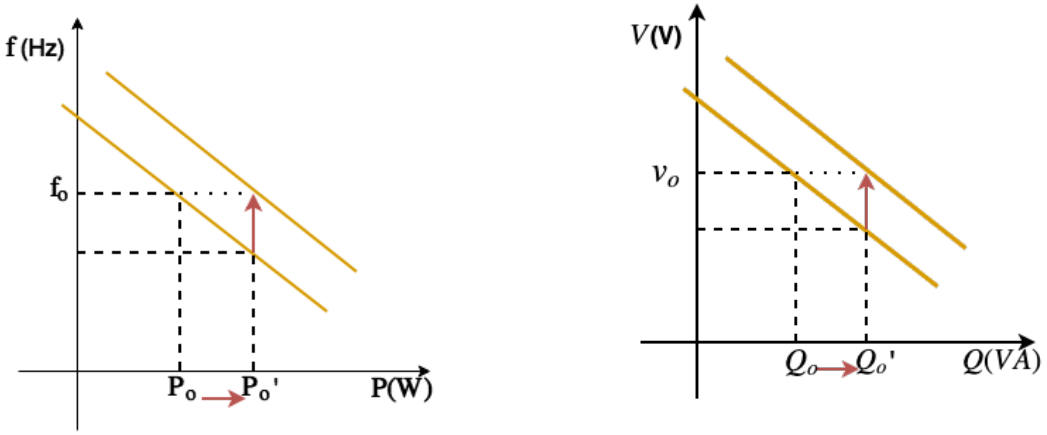


Figure 3: Diagram of the secondary control implemented in each VSI of the application support.

The principle of the secondary control is to shift the droop characteristic by changing the initial power values, as presented in Figure 4.



(a) f/P. (b) V/Q.

Figure 4: Droop characteristics [17].

The new initial values of active and reactive power P'_o and Q'_o are set based on (9):

$$\begin{cases} P'_o = P_o + \Delta P \\ Q'_o = Q_o + \Delta Q \end{cases} \quad \text{Considering that:} \quad \begin{cases} -m\Delta P = \Delta f \\ -n\Delta Q = \Delta V \end{cases} \tag{9}$$

With m and n the droop coefficients of the active and reactive powers respectively. Without knowing the dynamics of the control of the VSI, the variability of frequency and voltage magnitudes are computed using (10).

$$\begin{cases} \dot{f} = \dot{f}_o - m(\dot{P}_o - \dot{P}) \\ \dot{V} = \dot{V}_o - n(\dot{Q}_o - \dot{Q}) \end{cases} \tag{10}$$

In the secondary control, the set-point frequency (f_o) and the set-point voltage magnitude (V_o) are constant. However, the initial active and reactive power vary by a quantity ΔP and ΔQ respectively. So, (10) becomes:

$$\begin{cases} \dot{f} = -m(\dot{P} - \dot{\Delta P}) \\ \dot{V} = -n(\dot{Q} - \dot{\Delta Q}) \end{cases} \tag{11}$$

Combining (9) and (11), we can write:

$$\begin{cases} \dot{\Delta f} = \dot{f} + m\dot{P} \\ \dot{\Delta V} = \dot{V} + n\dot{Q} \end{cases} \tag{12}$$

The objective of the secondary control is finally to provide the set-points of Δf and ΔV to the primary control when the frequency (`Freq`) and the voltage magnitude (`Volt`) of the system differ from their initial reference. To ensure an effective active power sharing, the product of the active power and the droop coefficient of each generator must be identical [20]. This is expressed in (13).

$$m_i.P_i = m_j.P_j \tag{13}$$

The *synchronous tracking error* method allows synchronizing the frequency, the voltage and the active power sharing of all DER-clusters [21]. It defines the derivation of the frequency, voltage and active power sharing of each generator as:

$$\begin{cases} \dot{f}_i = c_f \times e^i_f \\ \dot{v}_i = c_v \times e^i_v \\ m_i.\dot{P}_i = c_p \times e^i_p \end{cases} \tag{14}$$

Where c_f, c_v and c_p are control coefficients and e^i_f, e^i_v and e^i_p determines the sum of the *neighborhood's error* at each node, which is done with (15). In this paper, the coefficients c_f, c_p and c_v are chosen identical and called c-coefficient.

$$\begin{cases} e_f^i = \sum_{j \in N_i} a_{ij}.(f_j - f_i) + a_{i0}.(f_o - f_i) \\ e_v^i = \sum_{j \in N_i} a_{ij}.(V_j - V_i) + a_{i0}.(V_o - V_i) \\ e_p^i = \sum_{j \in N_i} a_{ij}.(m_j.P_j - m_i.P_i) \end{cases} \tag{15}$$

Where a_{ij} is a coefficient representing the connection of node i to node j (equal to 1 if connected and 0 if not).

The derivation of the reactive power is computed with the low-pass filter, expressed in (16) in the Laplace domain.

$$Q = \left(\frac{w_c}{w_c + s}\right) \times Q' \rightarrow sQ = w_c(Q' - Q) \equiv \dot{Q} \tag{16}$$

Where Q' and Q are feedback values of the reactive power, respectively before and after the low-pass filter. The cut-off frequency of the filter is w_c.

To implement this method in the real-time experimental installation, all equations have to be transformed in their discrete-time representation. The output is then calculated as expressed in (17):

$$\begin{cases} \Delta f_i = \int \dot{\Delta f_i} dt \\ \Delta V_i = \int \dot{\Delta V_i} dt \end{cases} \tag{17}$$

These Laplace functions are transformed in their discrete-time representation with the sample time T_s.

$$\begin{cases} \Delta f_i(k) = T_s u_f(k) + \Delta f_i(k-1) \\ \Delta V_i(k) = T_s u_v(k) + \Delta v_i(k-1) \end{cases} \quad \text{with} \quad \begin{cases} u_f(k) = -c(e_f^i(k) + e_p^i(k)) \\ u_v(k) = -c(e_v^i(k) - u_q(k)) \\ u_q = -n_i(Q(k) - Q(k-1))/T_s \end{cases} \tag{18}$$

The method converges if $(T_s \times c) < 1$ [21]. In order to evaluate the performance of the implementation, the c-coefficient is fixed, and T_s varies. Since T_s depends mainly on the sum of the computation and communication times. Thus, the value of T_s impacts the "smoothness" of the system response. The smaller T_s is, the "smoother" the system response is. On the contrary, if T_s is too big and violates the converged criteria, the system will be unstable and possibly go out rapidly of control.

4. Implementation with Smartlog

Smartlog is a declarative language of data manipulation which proceeds at a higher level of abstraction than imperative programming. That may be not familiar to programmers. In this part, through the implementation of the secondary control, we describe how to program in this new proposed language.

4.1. Programming in Smartlog

Smartlog aims to accomplish *a specific task* rather than following pre-defined steps of algorithms. *Smartlog* is also a reactive language. The computations are triggered by data updates. Thus, algorithms programmed in *Smartlog* are *incremental*. Whenever a piece of data changes, it attempts to save time by only recomputing the outputs that depend on the changed data.

The purpose of the secondary control is to adjust Δf and Δv to restore the frequency and voltage magnitude to a given set-point. In a distributed way, this problem is composed of two sub-problems: new measured data at the local node and new received data from other nodes.

The identification (ID) of each node is an integer number. The reference values of frequency and voltage are set to 50 Hz (standard frequency in Europe) and 311 V (nominal value of a three-phase low-voltage balanced system) respectively. The c-coefficient depends mainly on the network characteristic. It is chosen so that $T_s > T_{delay}$ (T_s should be superior to the delay due to the communications time T_{delay}), with in mind the condition of convergence of the implemented method. In the context of the presented implementation, this gives a c-coefficient of 2.

4.2. Structure of the database

- `Measure(ID, Freq, Volt, Pow, Rep)` holds the instantaneous measure data of the frequency ($Freq$), the voltage ($Volt$), the active power (Pow), and the reactive power (Rep) for each node;
- `Doop(ID, m, n)` defines the droop coefficients for each DER converter;
- `DynamicMeasure (ID, Freq, Volt, mPow, nRep, Rep)` describes the data sent to neighbor nodes;
- `Neighbor(ID, ID-Neighbor, Address)` defines the neighbor name/network address mapping;
- `NeighborData(ID, ID-NEIGHBOR,` δf_{ij}, e_f^i, δv_{ij}, e_v^i,δp_{ij}, e_p^i`)` provides the local neighborhood errors of frequency, voltage, and power sharing;
- `NeighborSum(ID,` $\sum \delta f_{ij}$, $\sum \delta V_{ij}$`)` sums the local neighborhood errors;
- `Output(ID,` Δf_i, ΔV_i`)` gathers the compensation values of frequency and voltage for each node;

The above described attributes in data_types are key attributes. In this experiment, the distributed program is written manually for each computing unit. The rules using the *Smartlog* language are expressed in Listing 10. This program is loaded into each computing unit of the real-time experimentation.

Listing 10: Distributed secondary control implementation in Smartlog.

```
Module(Measure){
//rule1
& DynamicMeasure(i,f,v,mp,nq,q) :- Measure(i,f,v,p,q), DynamicMeasure(i,_,_,_,_,qo), f<>50,
    Droop(i,m,n), mp:=m*p, nq:=n*(q-qo), Neighbor(i,j,@k);
}
Module(DynamicMeasure){
//rule2
NeighborData(i,j,ef,sef,emp,semp,ev,sev):- DynamicMeasure(i,f,v,mp,_,_),
    NeighborData(i,j,df,_,dmp,_,dv,_), DynamicMeasure(j,fj,vj,mpj,_,_), ef:=fj-f,
    sef:=-df+ef, emp:=mpj-mp, semp:=-dmp+emp, ev:=vj-v, sev:=-dv+ev;
//rule3
NeighborData(i,j,ef,sef,emp,semp,ev,sev):- DynamicMeasure(j,fj,vj,mpj,_,_),
    NeighborData(i,j,df,_,dmp,_,dv,_), DynamicMeasure(i,fi,vi,mpi,_,_), ef:=fj-fi,
    sef:=-df+ef, emp:=mpj-mpi, semp:=-dmp+emp, ev:=vj-vi, sev:=-dv+ev;
//rule4
Output(i,ef,ev,now) :- DynamicMeasure(i,f,v,_,nq,_), Output(i,efo,evo,to),
    NeighborSum(i,sf,sv), ef:=(efo+(now-to)*0.001*2*(sf+(50-f))),
    ev:=(evo+(now-to)*0.001*(2*(sv+311-vm)+nq)).
}
Module(NeighborData){
//rule5
NeighborSum(i,ef,ev):- NeighborData(i,_,_,sdf,_,smp,_,sdv), NeighborSum(i,df,dv),
    ef:=df+sdf+smp, ev:=sdv+dv.
}
```

When a data instance in `Measure` is modified and its frequency value violates the set-point value, the corresponding instance in `DynamicMeasure` is sent to the neighboring nodes (rule 1) and the neighborhood error is updated in the current node (rule 2). Otherwise, if a node receives the modification of neighbor's data, it also updates the neighborhood's error (rule 3). Every change in the neighborhood's error leads to the incremental update of the sum of the neighborhoods' errors (rule 5). The output control values change consecutively if there is a modification of each instance of measure data, itself or one of its neighbors (rule 4).

5. Experimentation and proof of concept

The objective of the experiment is to prove that *Smartlog* is capable of providing the same results as an imperative language, in a more efficient way, with advantages discussed in Section 6.

The chosen model is built and simulated in the Matlab/Simulink environment. Then, it is loaded in an OPAL-RT target for real-time execution. Raspberry Pis 2 Model B (a small ARM computer) oversee the local computing unit of each generator (i.e. each node). Each Raspberry Pi is implemented following the node architecture shown in Figure 1. PostgreSQL is used to manage the database and plays the role of the rule engine. The distributed *Smartlog* program is executed with this rule engine. A small dedicated java program is in charges of the interfaces. This experiment presents a distributed database network with three raspberry Pis corresponding to three generators. The measured data are collected in the OPAL-RT server and sent to the Raspberry Pis (the virtual OPAL-RT server is configured to exchange measured data and control signals with each Raspberry Pi). Simultaneously, each Raspberry Pi communicates with each other via a TCP/IP protocol and sends back two attributes of their output data to OPAL-RT for the control of the simulated isolated microgrid. As OPAL-RT and the system of Raspberry Pis are configured in the same network, the IP address supplies enough information to identify a node in the network.

The architecture of the real-time platform used for the experiment is presented in Figure 5. The communication time between OPAL-RT and the Raspberry Pis is estimated to be around 10 ms.

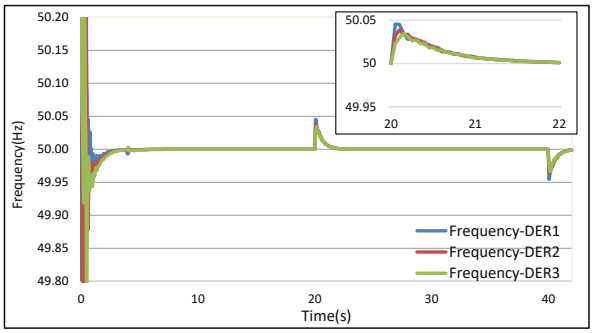
Figure 5: Architecture of the experimental installation.

We perform the simulation of the microgrid in isolated mode for 60 s. The load changes at T=20 s with an increase of 25% of the maximal power and comes back to normal at T=40 s. The parameters of the primary control are computed so that the stability of the grid, when the load changes, is reached within 0.5 s.

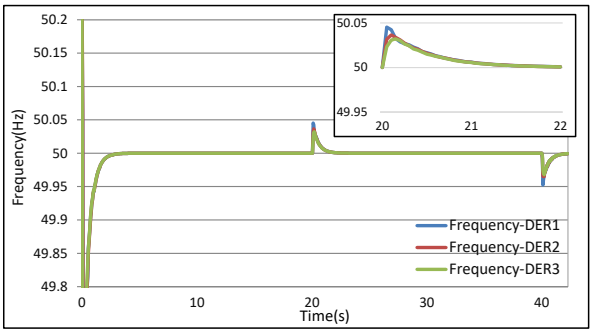
For comparison, the same implementation is done using Java JDK 1.8.0 (whose data are stored in the temporary memory, not in the database). Java was chosen as a good archetype of imperative language that fits well in the scope of the comparison with the proposed declarative language. Also, current multi-agent systems typically use Java (Python could be a possibility, but it is slower) and Java can be compiled in native mode on the system used for the experimentation (it could be based on C with a decreased ease of use). The Java implementation is used as a reference to validate the proper operation of the *Smartlog* implementation. The Java program is available in [22] as additional research data. The comparison of both implementations is also conducted to confirm the quantitative evaluations as well as the impact of the execution in the database on the response time with the *Smartlog* implementation.

Figure 6, 7 and 8 present the frequency, voltage and active power variations after the load change and the action

of both the primary and secondary controls with two implementations: one in *Smartlog* and another in Java.

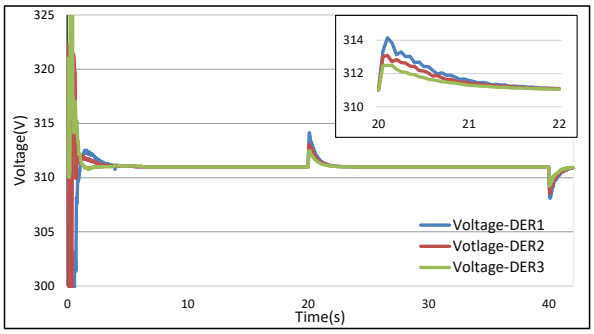


(a) In *Smartlog*.

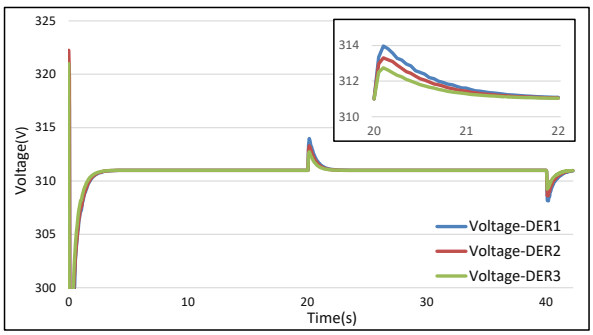


(b) in Java.

Figure 6: Frequency response after a 25 % load change and the action of both the primary and secondary controls.



(a) In *Smartlog*.



(b) In Java.

Figure 7: Voltage magnitude after a 25 % load change and the action of both the primary and secondary controls.

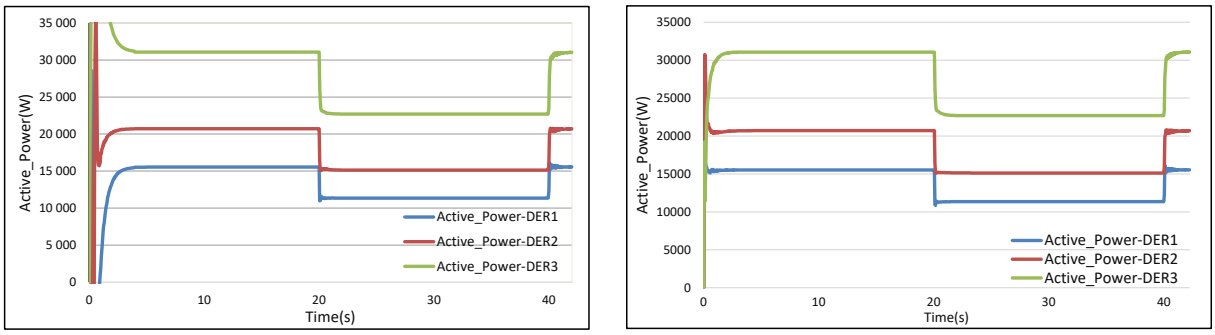
Figure 6a, 7a, and 8a show that the *Smartlog* implementation provides a satisfactory secondary control compared to the equivalent java implementation, which present similar response times (Figure 6b, 7b, and 8b). When the load changes, at T=20 s, and T=40 s, the frequency and the voltage oscillate slightly due to the regulation. The active power is also correctly shared between the three generators during the experiment. The results confirm that the distributed node architecture is appropriate for the *Smartlog* implementation, validating the convergence of the *Smartlog* program and finally illustrating the distributed programming mechanism of *Smartlog*.

6. Discussion: advantages and disadvantages of *Smartlog* compared with other languages

6.1. Smartlog compared with imperative languages

6.1.1. Simplicity

The *Smartlog* language is well adapted with database management and communication. It uses simple symbols (“^”, “@”) to describe communications where other languages need much more command lines. Besides, it is not



(a) In *Smartlog*.

(b) In Java.

Figure 8: Active power sharing after a 25 % load change and the action of both the primary and secondary controls.

necessary to declare variables in modules, because this declaration is done automatically and optimally. Through each module, a *Smartlog* program clarifies *which* sub-problem in the system is to be considered, so no unnecessary commands are executed.

Each rule in *Smartlog* represents a concrete problem. A rule can be a combination of the condition operator and computations, i.e. the execution. This combination leads to program being more perspicuous and closer to a natural language. The secondary control is programmed in only five rules in *Smartlog* (i.e. five lines of code). Moreover, the implementation in *Smartlog* is simple, because it works on the database which is available in each node in the smart grid. No sequence of code is really defined in the algorithm, as computations depend on the meaning of data, they do not only follow the steps of an algorithm. The output of the control is just adjusted after a change in the data.

Compared to the Java implementation presented in the experimentation of Section 5, similar responses times (with light differences of operation) provide an advantage to *Smartlog* which automatically distributes, in the form of rules, a control developed for a centralized implementation. It thus eases both the implementation, the operation and the bandwidth solicitation. As an illustration, the number of command lines in the Java implementation is quite significant compared to the *Smartlog* implementation. The compactness of *Smartlog* regarding the length of the implemented programs in this test-case is summarized in the comparison of Table 1.

Table 1: Secondary control implemented in Smartlog and Java.

Features	Smartlog	Java
Number of lines of code	5	54
Number of instructions	36	108

The lower number of instructions in the *Smartlog* language to implement the same algorithm does not mean that *Smartlog* instructions present a better performance than the implementation in Java. Actually, the number of machine-level instructions is ideal for this comparison but was not easily accessible for this experiment, because both the Java

and the *Smartlog* implementations were not directly compiled to machine code. Thus, even if the number of code lines and instructions do not really reflect the performance of the program, the fact that the implementation in *Smartlog* needs almost 10 times less lines of code and 3 times less instructions than the implementation with Java provides a relevant first comparison.

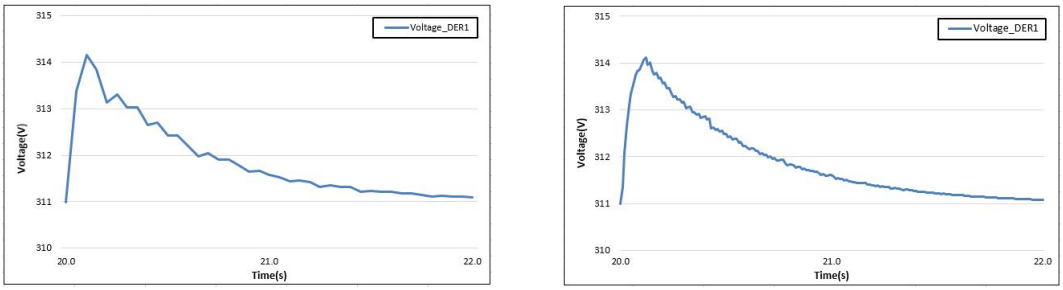
6.1.2. Scalability and capacity to address an incremental volume of data

Smartlog is designed for declarative programming. The computation is performed based on detecting changes in the database. *Smartlog* supports algorithms developed for real-time control, based on the current measured data as well as historic data. The output values are corrected step by step. For that reason, no sample time or communication time between neighbors are fixed. Those timings are defined by the interval between updates occurring in data and the actual computation time. Moreover, by keeping the same configuration of the network and the same program (modules), the performance of the computation is conducted on an increasing number of Raspberry Pis. We get the same result, but the computation is shared. That proves the potential of the approach in sharing the calculations and data as the system of computing resources grows.

6.1.3. Impact of execution in the local database on computing time

Smartlog is designed to perform computations distributed in local databases. This architecture presents some advantages but affects the performance of the program's execution. This particular application is also interesting to evaluate the possibilities of control and management of a microgrid with *Smartlog*.

As stated in Section 3.3, the sample time T_s affects the smoothness of the response. When we look more closely at the voltage response of the first generator (at T=20 s) with both implementations, presented in Figure 9, the Java implementation presents less variations.



(a) With *Smartlog*.

(b) With Java.

Figure 9: Voltage response after a load change for one of the generators of the microgrid.

It means that T_s is bigger with the *Smartlog* implementation than with Java one. This can be explained simply by the fact that data in the Java implementation are stored in Read-Only Memory, which has a faster query time than the database used in the *Smartlog* implementation. However, it should be admitted that this does not affect the results and

the response time of the system significantly. The procedure for the acceleration of computing time in *Smartlog* can be done, for example, by using in-memory databases or dedicated rule execution engines.

6.2. Smartlog comparison with declarative languages

To the best of our knowledge, there is no declarative language fully operational in the context of smart grids. Either there is no available compiler, the database management is lacking, or, like for example in the case of Datalog, there is no real support for communication. *Smartlog* is filling this gap from our point of view.

7. Conclusion

In this paper, the abstract structure of all nodes in a smart grid's distributed control perspective, as well as the structure of a new declarative language, called *Smartlog* are proposed and illustrated on typical examples. The *Smartlog* language is used to program the implementation of a distributed secondary control in a microgrid in isolated mode. The code in *Smartlog* is also proposed and explained.

An experimental test is put up on a real-time OPAL-RT target connected to three Raspberry Pis. A program is written in a common imperative language (Java) for comparison. For the same control, *Smartlog* demonstrates its simplicity, compactness and incremental operation allowing it to better adapt to a large number of nodes in real-time. However, there are still work to be done to fully support its claim of scalability.

As future prospective work, the accelerating of data query time in *Smartlog* presents an interest. The automatic distribution of rules, as well as the database definition for smart grids can also be optimized. The implementation of other microgrid controls should be performed in *Smartlog* to prove the reduction of communication and computation time in this declarative paradigm. Finally, the scalability of the proposed language is also to experimentally validate.

Acknowledgment

This research was made possible by the funding provided by the *French Embassy in Vietnam* as well as the *Fondaction Grenoble INP* in France.

References

[1] R. Zafar, A. Mahmood, S. Razzaq, W. Ali, U. Naeem, K. Shehzad, Prosumer based energy management and sharing in smart grid, Renewable and Sustainable Energy Reviews 82 (2018) 1675–1684.

[2] G. Van de Kaa, T. Fens, J. Rezaei, D. Kaynak, Z. Hatun, A. Tsilimeni-Archangelidi, Realizing smart meter connectivity: Analyzing the competing technologies power line communication, mobile telephony, and radio frequency using the best worst method, Renewable and Sustainable Energy Reviews 103 (2019) 320–327.

[3] N. Shaukat, S. Ali, C. Mehmood, B. Khan, M. Jawad, U. Farid, Z. Ullah, S. Anwar, M. Majid, A survey on consumers empowerment, communication technologies, and renewable generation penetration within smart grid, Renewable and Sustainable Energy Reviews 81 (2018) 1453–1475.

[4] K. L. López, C. Gagné, M.-A. Gardner, Demand-side management using deep learning for smart charging of electric vehicles, IEEE Transactions on Smart Grid 10 (3) (2018) 2683–2691.

[5] A. Sanchez, W. Rivera, Big data analysis and visualization for the smart grid, in: Int. Cong. on Big Data, IEEE, 2017, pp. 414–418.

[6] C. A. Hans, P. Braun, J. Raisch, L. Grüne, C. Reincke-Collon, Hierarchical distributed model predictive control of interconnected microgrids, IEEE Transactions on Sustainable Energy 10 (1) (2018) 407–416.

[7] L. Wen, K. Zhou, S. Yang, L. Li, Compression of smart meter big data: A survey, Renewable and Sustainable Energy Reviews 91 (2018) 59–69.

[8] T. Yang, 10 - ict technologies standards and protocols for active distribution network, in: Q. Yang, T. Yang, W. Li (Eds.), Smart Power Distribution Systems, Academic Press, 2019, pp. 205 – 230. doi:https://doi.org/10.1016/B978-0-12-812154-2.00010-9.
URL http://www.sciencedirect.com/science/article/pii/B9780128121542000109

[9] W. Liu, W. Gu, Y. Xu, Y. Wang, K. Zhang, General distributed secondary control for multi-microgrids with both PQ-controlled and droop-controlled distributed generators, IET Generation, Transmission & Distribution 11 (3) (2017) 707–718.

[10] M. Pau, E. Patti, L. Barbierato, A. Estebsari, E. Pons, F. Ponci, A. Monti, A cloud-based smart metering infrastructure for distribution grid services and automation, Sustainable Energy, Grids and Networks 15 (2018) 14 – 25, technologies and Methodologies in Modern Distribution Grid Automation. doi:https://doi.org/10.1016/j.segan.2017.08.001.
URL http://www.sciencedirect.com/science/article/pii/S2352467716301783

[11] A. Ronca, M. Kaminski, B. C. Grau, B. Motik, I. Horrocks, Stream reasoning in temporal datalog, in: Thirty-Second AAAI Conference on Artificial Intelligence, 2018, pp. 1–8.

[12] H. M. Demoulin, T. Vaidya, I. Pedisich, B. DiMaiolo, J. Qian, C. Shah, Y. Zhang, A. Chen, A. Haeberlen, B. T. Loo, et al., Dedos: Defusing dos with dispersion oriented software, in: Proceedings of the 34th Annual Computer Security Applications Conference, ACM, 2018, pp. 712–722.

[13] S. Grumbach, F. Wang, Netlog, a rule-based language for distributed programming, in: Practical aspects of declarative languages, Lecture Notes in Computer Science, Springer, Berlin, Heidelberg, 2010, pp. 88–103. doi:10.1007/978-3-642-11503-5_9.

[14] A. Vaccaro, I. Pisica, L. Lai, A. Zobaa, A review of enabling methodologies for information processing in smart grids, International Journal of Electrical Power & Energy Systems 107 (2019) 516 – 522. doi:https://doi.org/10.1016/j.ijepes.2018.11.034.
URL http://www.sciencedirect.com/science/article/pii/S0142061518303405

[15] A. A. de Sotomayor, D. D. Giustina, G. Massa, A. DedÃ¨, F. Ramos, A. Barbato, Iec 61850-based adaptive protection system for the mv distribution smart grid, Sustainable Energy, Grids and Networks 15 (2018) 26 – 33, technologies and Methodologies in Modern Distribution Grid Automation. doi:https://doi.org/10.1016/j.segan.2017.09.003.
URL http://www.sciencedirect.com/science/article/pii/S2352467716302077

[16] T. T. Q. Nguyen, C. Bobineau, V. Debusschere, Q. H. Giap, N. Hadj-Said, Using declarative programming for network data management in smart grids, in: Proceedings of the 22Nd International Database Engineering & Applications Symposium, IDEAS 2018, ACM, New York, NY, USA, 2018, pp. 292–296. doi:10.1145/3216122.3216160.
URL http://doi.acm.org/10.1145/3216122.3216160

[17] A. Mehrizi-Sani, Chapter 2 - distributed control techniques in microgrids, in: M. S. Mahmoud (Ed.), Microgrid, Butterworth-Heinemann, 2017, pp. 43 – 62. doi:https://doi.org/10.1016/B978-0-08-101753-1.00002-4.
URL http://www.sciencedirect.com/science/article/pii/B9780081017531000024

[18] N. Pogaku, M. Prodanovic, T. C. Green, Modeling, analysis and testing of autonomous operation of an inverter-based microgrid, IEEE Trans. on Power Electronics 22 (2) (2007) 613–625.

[19] Ã“scar LucÃa, E. Monmasson, D. Navarro, L. A. BarragÃ¡n, I. Urriza, J. I. Artigas, Chapter 29 - modern control architectures and implementation, in: F. Blaabjerg (Ed.), Control of Power Electronic Converters and Systems, Academic Press, 2018, pp. 477 – 502. doi:https://doi.org/10.1016/B978-0-12-816136-4.00030-0.
URL http://www.sciencedirect.com/science/article/pii/B9780128161364000300

[20] W. Gu, G. Lou, W. Tan, X. Yuan, A nonlinear state estimator-based decentralized secondary voltage control scheme for autonomous microgrids, IEEE Trans. on Power Systems 32 (2017) 4794–4804.

[21] X. Lu, X. Yu, J. Lai, J. M. Guerrero, H. Zhou, Distributed secondary voltage and frequency control for islanded microgrids with uncertain communication links, IEEE Transactions on Industrial Informatics 13 (2) (2017) 448–460.

[22] N. T. Quynh, Implementation a distributed secondary control (May 2018). doi:10.5281/zenodo.1246756.
URL https://doi.org/10.5281/zenodo.1246756

Biographies

Thi-Thanh-Quynh Nguyen is currently a PhD student of Grenoble Institute of Technology. Her thesis is rolling in the collaboration of two laboratories: Grenoble Electrical Engineering Laboratory (G2Elab) and Grenoble Informatics Laboratory (LIG). Her research interests include Big data, distributed data management for smart grids and distributed control and management in microgrid.

Vincent Debusschere has obtained his Ph.D. in ecodesign of electrical machines from the Ecole Normale Superieure de Cachan in 2009. He joined the Electrical Engineering Laboratory of the Grenoble Institute of Technology, in 2010 as an Associated Professor. His research interests include renewable energy integration, modeling of flexibility levers for smart grids, multi-criteria assessment and optimal design of complex systems.

Christophe Bobineau has obtain his PhD in computer science from the University of Versailles Saint-Quentin in 2002. He is working since then at the Grenoble Informatics Laboratory (Grenoble Institute of Technology). His topics cover transaction management, distributed query optimization and data storage from embedded systems to Big Data.

Quang Huy Giap is a Lecturer-researcher at the University of Science and Technology, The University of Danang. He received his PhD in Automation-Production in 2011 from Grenoble INP, France. His research is situated in the field of Automation, with a special focus on fault detection and diagnostics technologies, renewable energies and energy management.

Nouredine Hadjsaid is a Professor at Grenoble Institute of Technology, Director of the Laboratory of Electrical Engineering of Grenoble (G2ELAB). He is also a visiting professor at Virginia Tech (USA) and NTU (Singapore). His research interests are in smart grids, which include distributed generation and power grids, information and communication technologies in power grids, and power grid safety, among others.