Une nouvelle approche pour la programmation distribuée dans les smartgrids

Thi Thanh Quynh Nguyen

Communauté UNIVERSITÉ Grenoble Alpes

THÈSE

Pour obtenir le grade de

DOCTEUR DE LA COMMUNAUTÉ UNIVERSITÉ GRENOBLE ALPES

Spécialité : **Génie Electrique**

Arrêté ministériel : 25 mai 2016

Présentée par

Thi Thanh Quynh NGUYEN

Thèse dirigée par **Nouredine HADJSAID**
et Co-encadrée par **Christophe BOBINEAU, Vincent DEBUSSCHERE, Quang-Huy GIAP**

préparée au sein du **Laboratoire de Génie Electrique de Grenoble et du Laboratoire d'Informatique de Grenoble**
dans **l'Ecole Doctorale Electronique, Electrotechnique, Automatique, Traitement du Signal (EEATS)**

A new approach for distributed programming in smart grids

Thèse soutenue publiquement le **16 Décembre 2019**,
devant le jury composé de :

Madame Marie-Christine ROUSSET
Professeure, Université Grenoble Alpes, Présidente
Monsieur David EDWARD BAKKEN
Professeur, Université Washington State, Rapporteur
Monsieur Vasile-Marian SCUTURICI
Maître de conférences (HDR), INSA Lyon, Rapporteur
Monsieur Nouredine HADJSAID
Professeur, Grenoble INP, Directeur de thèse
Monsieur Christophe BOBINEAU
Maître de conférences, Grenoble INP, Co-encadrant de thèse
Monsieur Vincent DEBUSSCHERE
Maître de conférences (HDR), Grenoble INP, Co-encadrant de thèse
Monsieur Quang-Huy GIAP
Maître de conférences, Université de Danang, Co-encadrant de thèse

Acknowledgements

The three years of the thesis are one of the most memorable periods in my life. I would like to express my heartfelt gratitude to all those who have contributed to the accomplishment of this work.

First of all, I want to express my sincere gratitude to my supervisor, Professor Nouredine HADJSAID, for welcoming me to work in G2Elab and his valuable advice throughout the research. Besides, I would also like to thank my supervisor in Vietnam Mr.GIAP Quang-Huy, from Danang university of technology, always encourage me as well as be willing to supply useful data of Vietnam electrical grid. Particularly, I would like to express my deep and sincere gratitude to my two supervisors, Mr. Christophe BOBINEAU from LIG and Mr. Vincent DEBUSSCHERE from G2Elab, who kindly supervised during my thesis. Without their precious support, their advice, their encouragement and their spirit of tolerance, I could not have completed this research work.

I would like to thank Mr. David E.BAKKEN and Mr. Vasile-Marian SCUTURICI for accepting to be the rapporteurs of my thesis. I am grateful to them for all the time they have devoted to the reading and correction of my manuscript, and for their enriching remarks. I also express my sincere thanks to Ms. Marie-Christine ROUSSET for accepting the responsibility of the examiner.

I also thank Mr. Antoine LABONNE, Mr. Cédric BOUDINET, and Mr. Tung-Lam NGUYEN for assistance and advice. They taught me a lot about RT-LAB® and the power grid models. Without their help, I could not have finished my experiments.

Special thanks to Mr. NGUYEN Khac Nhan, who introduced me to Professor Nouredine. His sense of responsibility and his great availability have given me valuable support and a source of motivation. I am also grateful to Ms. VO Thi Hoang Chau and Mr. HO Phuoc Tien of the Danang University for recommending me to Professor Nhan.

I would like to thank all the staffs of the administrative services, computer services of the two laboratories G2Elab and LIG, for having supported me during these 3 years of the thesis. I really love the working environment in these two laboratories.

I express my gratitude to the Embassy of France in Vietnam and the Grenoble INP Foundation for their valuable financial support.

I would also like to thank all my colleagues from both laboratories especially the HADAS and SYREL teams and Vietnamese friends in France for their friendship and supports. I will never forget the good time together. You had warmed my heart and dispelled the feeling of loneliness while I was away from my family.

Finally, I am deeply thankful to all my family members, my boyfriend, especially to my grandmother, for their endless love and their unconditional supports. All of you are my solid rear, you have given me strong faith overcome all difficulties. I'm grateful to have you all in my life.

Thi-Thanh-Quynh NGUYEN
Grenoble, France

Contents

List of Figures

List of Tables

General introduction

The world is facing global climate change. The average temperature of the earth has increased more than 1.5°C in the past decade. One of the principle reasons causing global warming is the greenhouse effect and CO_2 equivalent emissions, more than 30% [16] of which are due to electricity production. Besides, the "Industry 4.0" revolution requires significant electrical power for the operation of electrical devices, increasing even more the environmental impact of the sector.

Most countries in the world, for instance Vietnam, are going to cope with the lack of energy in the future. In that context, the transformation of usage from energy resources towards cleaner fuel is an inevitable trend. However, the interruption, as well as the scattered distribution of these resources, pose a serious issue: How to integrate them into the power grids while ensuring the stability of these systems?

The smart grids and microgrids concepts appeared and are being developed, notably, as one of the answers to those issues. Combining the power grid and the Information and Communication Technology (ICT) allows an active and more autonomous control of the power systems, and thus makes them more resilient, durable, and robust. We call the information network in this combination an Active Management Network (ANM), in which a large number of sensors, smart devices, and actuators intervene.

The main challenge of the smart grid is its capability to deal with the big volume of data generated by a vast number of sensors (data generators). We can easily deploy conventional algorithms executed using centralized paradigm (like in the traditional power grids). Nevertheless, there are enormous volumes of data collected by the systems, but there are only few grid operators analyzing the data. The rush of data at several points as well as the huge data traffic in a telecommunication network with limited bandwidth may lead to system failures (e.g., bottleneck phenomenon, reduction in data velocity,

single-point failure). This problem becomes severe as the system scales up. In that situation, the smart grid needs developments to transform from a centralized paradigm to a distributed one. The distributed paradigm can deal with the scalability issue of the system by allowing to share computations by making use of all available computing devices. Moreover, data are processed "in-network" (i.e., near the data producers and consumers), which reduces the possibility of data congestion at the supervisor gate and the risk of single-point failure. This paradigm is fully adapted to control applications in microgrids and multi-microgrids [37].

Current researches on distributed control still show many drawbacks. Indeed, there is no clear mechanism dedicated to handle sensing data (e.g., history data, instantaneous data) in distributed computing units. The client-server paradigm supports distributed computing in which data are stored centrally. That is not convenient for the distributed data in smart grids. Besides, the distributed algorithm to deploy traditional electrical applications in a distributed memory system is still not practical. For example, *consensus* algorithms are used to achieve agreement on a single data value among distributed processors or systems [99]. The convergence of the algorithm depends mainly on the network configuration, and thus lacks programming abstraction because each node has its own program with different connection weight. That potentially impacts the distributed implementation when the system evolves. In addition, the Alternating Direction Method of Multipliers (ADMM) [24] is suggested as a decomposition approach for optimal power flow because its convergence is faster than other decomposition approaches such as Gauss-Seidel Relative Error (GS-RE) [7], the Diagonal Quadrative Approximation (DQA) [84]. However, it takes many communications (up to thousands) to reach a convergence point.

This thesis is placed in the context of the evolution of the traditional power grid towards smart grids. Through the stormy development and integration of the information network, smart grids promise a full control and management, even at the lowest level of the power grid. We aim at developing distributed control for smart grids. One of the most challenging tasks of the distributed paradigm is programming. It needs to handle a fine-level detail of the synchronization and message exchange among participants, but a high-level programming abstraction is still lacking. We propose to focus on data and their manipulation instead of processing distributed programming: we abstract the whole smart grid as a distributed database, and thus we can use a declarative

approach to express distributed data manipulations. A distributed database is a set of cooperating local databases (one per participating computing device) that perform local data manipulations and exchange data for synchronization. The declarative approach for data manipulation aims at describing the expected results rather than the detailed step of computations.

Supervision and control of smart grids is reactive in its essence, i.e. the system must react as soon as possible to abnormal situations identified by sensors. So, a declarative and reactive approach will be more appropriate. Data manipulation languages using this approach are based on first-order logic. A program is then a set of rules triggered by a modification in the dataset and may produce new data that probably trigger other rules or actions (via actuators). Datalog was the first proposal of such a logic-based language but runs only in central databases. Datalog extensions were proposed, such as NDlog or Netlog, that enable distributed execution by introducing communication primitives. However, these languages are not exactly adapted to smart grids. Since there is no key concept in data storage; there is no support for extended functions, and there is no control of the execution order of rules. Therefore, we develop a new logic-based distributed data manipulation language, called *Smartlog* to fill these gaps.

Even if *Smartlog* simplifies distributed programming in smart grids, we still have to manage synchronization and data exchange among participating computing devices. So we propose to develop a new methodology of distributed programming in *Smartlog*. The principle of this methodology is to program centralized algorithms in Smartlog, because there is a strong set of existing algorithms for traditional grids, but to execute them in a distributed manner, because it is more efficient and scalable. We call this methodology CPDE, for Centralized Programming and Distributed Execution. To do this, centralized *Smartlog* programs have to be rewritten into distributed ones according to data allocation.

In this thesis, we address the problem of data management and algorithm deployment in a heterogeneous network like a smart grid. This thesis aims to propose a new implementation approach for the distributed paradigm dedicated to smart grids. The major contributions of this thesis are summarized as follows, based on chapters descriptions:

1. We investigated the problem of algorithm deployment in smart grids and exposed

the limitations of more conventional methods which deal with a distributed implementation (Chapter 1).

2. We proposed to abstract smart grid management as data manipulation in distributed databases so that each smart device can manage and enhance the stability and accuracy of its local data and we develop a high-level declarative data manipulation language, called *Smartlog*, dedicated to smart grid needs. The main strengths of this language are its reactivity, recursiveness, simplicity, ease of implementation, and good support for distributed computations. We would like to emphasize that declarative approaches have never been used before in electrical engineering to the best of our knowledge.

3. We designed the node architecture for the distributed database system, implemented using PostgreSQL and its triggers as a local database and a rule execution engine respectively. We also developed a *Smartlog* compiler for this prototype. The detail of these contributions is presented in Chapter 2.

4. We developed a methodology for *Smartlog* programming, called CPDE, that supports semi-automatic distributed programming in *Smartlog*. This approach helps execute centralized algorithms in distributed systems. Details of this method are presented in Chapter 3.

5. We conducted several experiments on a real-time simulation platform (real-time simulation machine OPAL-RT and a network of Raspberry Pis as computing devices), in both implementations: declarative programming (*Smartlog*) and imperative programming (JAVA) to compare and evaluate our development. The qualitative evaluations through four typical applications for smart grids are presented in Chapter 4.

6. We evaluated the performance of distributed *Smartlog* execution. For the last application, the performance of the distributed *Smartlog* execution is compared to the centralized one and to a reference Java one. Besides, the aspects which affect the performance are studied. The detail of performance evaluations is shown in Chapter 5.

Finally, we summarize our works and propose further research directions in the conclusion chapter.

Chapter 1

Background

1.1 Context of the work

1.1.1 Global warming and possible lack of energy in the future

According to the Intergovernmental Panel on Climate Change (IPCC), the instrumental temperature record shows a global warming of around 1.5°C during the 20^{th} century [70]. The temperature will continue to rise in the future, according to Representative Concentration Pathways (RCPs) scenarios shown in Figure 1.1. Each scenario corresponds to effort to constrain CO_2 emissions, with RCP 8.5 is a very high emission scenario. The global warming contributes to the sea level rise, and the increasing frequency of disasters such as storms surges and hurricanes in coastal areas. All that represents a clear risk to our environment. That also entails the escalation of electrical demand for cooling systems.

The greenhouse effect and a large amount of CO_2 emissions are the main cause of global warming. As the report by BP statistical review of world energy in 2018, 34.7 % of the CO_2 emitted from global fossil fuel combustion in 2017 came from electricity generation [16]. Among that, China and India are two leading countries regarding CO_2 emissions since the 80s. On the 12th of December 2015, the United Nation Climate Change conference COP 21 was held in Paris with the participant of 196 countries. This conference achieved a global agreement to reduce emissions of greenhouse gases. Besides, the increase of the price of fossil fuels in the world and serious considerations about the

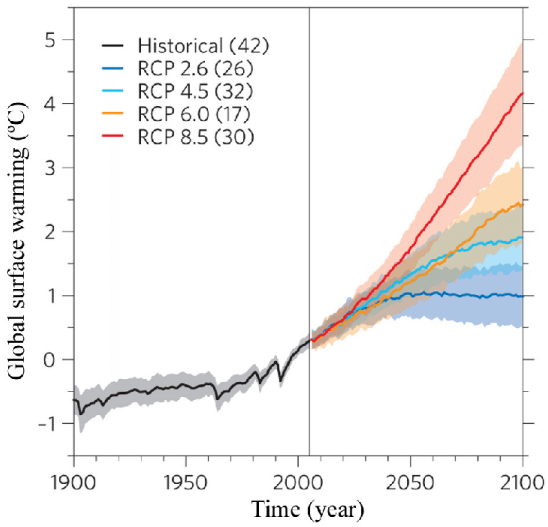


Figure 1.1: Global temperature increase at the surface ot the earth [70].

safety of nuclear power plants, which re-emerged after the Fukushima disaster in 2011 in Japan, caused concerns about the future energy developments.

The development of the "industry 4.0" requires a great energy demand. In the world, the global electricity production grows on average of 3 % annually. Meanwhile, the global electricity demand is expected to increase by around 25 % from 2016 to 2040 [4]. 45 % of this electrical demand growth will be contributed by China and India. In Vietnam, the electricity production increased sharply, 12 times from 1995 to 2018. According to the report of EVN [95], the electricity demand in Vietnam is now bigger than the production and the increase of electricity demand is about 10.7 % annually.

In terms of production, the energy resources will shift towards cleaner fuels such as renewable energy (wind, solar) or natural gas. In 2017, renewable power accounted for 70 % of the net additions to the global generating capacity, raising the total capacity by 8 % over 2016. Among that, investments in solar production have increased by 18 %. Costa Rica, Belize and Austria, with a percentage of renewable energy in the total energy system of 98 %, 91 %, and 73 % respectively, are the current leading countries in renewable resource usage [81].

1.1.2 Focus: renewable energy production in Vietnam

The developments of this PhD thesis are targeting developing countries like Vietnam. There, the development of renewable energy has a great potential. After canceling

the plan to build nuclear power plants, the Vietnamese government encouraged the development of renewable energy. According to the electricity 7^{th} plan [14], at the moment, there are 245 renewable energy projects signed, which promises to supply electricity power up to 23.2 GW in 2020 [21]. The Vietnamese government will support tariffs for renewable energy purchasing. In detail, the tariff is 8.77 US cent/kWh for on-shore wind power projects, and 9.97 US cent/kWh for off-shore wind power projects [93]. For solar energy, the tariff is 9.35 US cent/kWh. These tariffs are quite high in comparison to the current tariffs in France (7.96 US cent/kWh for solar for example) [49]. The solar energy is a part of the renewable energy development project of the Vietnamese government. According to [94], the goal is to increase the percentage of renewable energy resource specially in solar energy to 6 % in 2030 and 20 % in 2050. Up to July 2018, there is 748 projects for solar energy deployed over the country with the power promising to reach 11.5 MWp in 2050.

1.1.2.1 Potential of solar energy in Vietnam

As a tropical country, there is a relatively high and stable resources each year. On average, the national solar energy is 4 to 5 kWh/m^2 per day, as presented in Table 1.1.

Table 1.1: Theoretical potential of solar power in Vietnam [73].

Zone	**Average total irradiation** (kWh/m^2/day)	**Surface** (km^2)	**Potential power** (MWh/day)
Northeast and Red River Delta	3.95	65.6	21,100
Northwest region	4.80	50.7	19,800
North central	4.90	51.5	20,500
South Central and Highlands	5.30	99.0	42,700
Southeast and Mekong Delta	5.15	64.2	26,900
Total		331	131,000

According to Vietnam Green Innovation and Development (GreenID), the potential of solar power exploitation in Vietnam is 13,000 MW [44]. But up to now, the total capacity of installed solar panels for production is only about 5MW.

1.1.2.2 Potential of wind energy in Vietnam

According to [89], and presented in Figure 1.2, the east sea, northeast and southwestern regions, present average wind speeds of 10 to 11 m/s. The sea area from Binh Thuan to

Ca Mau, 300 km from the coast, presents average wind speeds of 7 to 11 m/s. It is also the world's largest wind energy potential.

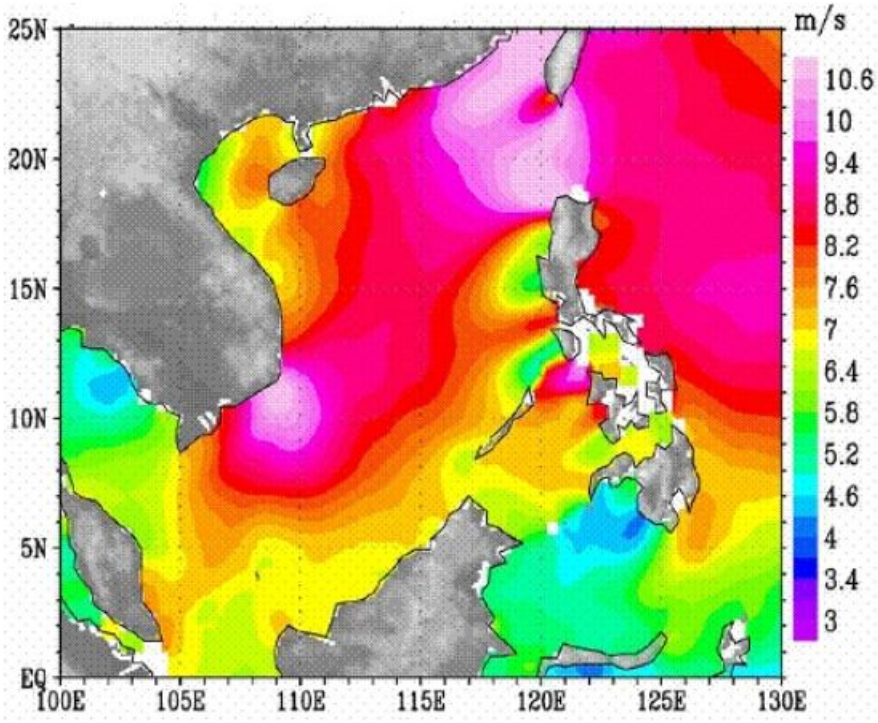


Figure 1.2: The potential of offshore wind power in Vietnam, 2000–2009 [89].

The potential of offshore wind power in Vietnam could reach to 64.8 GW with depths up to 30 m and 107 GW with depths from 30 to 60 m, as summarized in Table 1.2 and Table 1.3.

Table 1.2: Potential of offshore wind power, depth 0–30m [89].

Zone	**Surface** (km^2)	**Wind energy density** (MW/km^2)	**Potential power** (GW)
Gulf of Tonkin	30.8	400	12.3
Quang Binh - Quang Ngai	4.66	500	2.33
Binh Dinh - Ninh Thuan	2.48	500	1.24
Binh Thuan - Mui Ca Mau	43.8	850	37.2
Mui ca Mau - Kien Giang	29.4	400	11.8
Total	111		64.8

To conclude, Vietnam has a huge potential for renewable energy. The development of these energy sources integrated into the grid system is essential to meet the increasing energy demand. Moreover, it contributes to reducing global warming based on its primary resource.

Table 1.3: Potential of offshore wind power, depth 30–60m [89].

Zone	**Surface** (km^2)	**Wind energy density** (MW/km^2)	**Potential power** (GW)
Gulf of Tonkin	29.2	550	16.1
Quang Binh - Quang Ngai	7.10	500	3.55
Binh Dinh - Ninh Thuan	2.11	500	1.06
Binh Thuan - Mui Ca Mau	68.0	1000	68.0
Mui ca Mau - Kien Giang	36.0	500	18.0
Total	142		107

1.1.3 Evolution and challenges of the power grid

The electricity is still costly to store. At any time, the quantity of electricity demanded (plus losses) must be equal to the quantity of power injected into the grid to avoid blackouts. In current grids, electricity flows mainly in a single direction, from producers to customers, and information from customers to producers. But in the future, grids should evolve to handle a lot more constraints and bidirectional flows of both energy and information.

Integration of variable renewable energy resources: With the integration of variable renewable energy resources (VRES), mostly connected through power electronics, occurs a reduction of the inertia of the grid (its capacity to damp variations of electrical quantities). Thus, the development of renewable energy resources is usually restrained by the stability limits of the current grids. Besides, those resources are often allocated scattered in the grid and requiring energy transportation. That leads to an increasing complexity for the energy system.

Augmentation of the number of connected devices: The "industrial revolution 4.0" expects an increment in the number of electrical devices which will cause a growing electrical consumption. In addition, the emergence of new uses, such as electric vehicles, will amplify this rise. Along with this foreseeable increase, the current power grids will no longer be sufficiently adapted. To avoid reinforcing grids, which is expensive, it will be necessary to control those loads and extract flexibility from them; for example, by optimizing the charging periods of electric vehicles [22].

Energy management and two-way communication: The balance between the supply and the demand cannot follow the traditional one-way (from the producer to the customer) because, in smart grids, both the producer and the customer participate in the power management to maximize their benefices and increase the possibility to cope with VRES. In a two-way interaction, the communication network will have more importance, as well as in its interaction with the energy grid.

Resilience: The scalability of energy resources should lead to a change in architecture. Particularly in Vietnam, the centralized architecture in its current situation shows lots of drawbacks. For instance, an accident in a 500 kV line caused a blackout in a wide area on the 22^{nd} of May 2013. It has lead to 19 electrical factories and 43 generator sets to stop operating. The sum of the power blackout was of about 9,400 MW. More than 1.8 million citizens in the south area had no electricity [98].

Moreover, the Vietnam presents a "S-shape", spreading on 1,650 km long and about 50 km wide at the narrowest point. That would require a decentralized production in the future as well as renewable energy developments for the long term to ensure an electrical security of supply, whereby, the small electrical factory is scattered throughout the territory. Each zone has to be as much independent from the supply as possible in the electricity, but there is still a physical connection for back-up with other zones in case of a severe incident. The rest of the time, in each zone, it should be balanced between supply and demand to avoid a potential cause of blackout. These constraints in the power systems could be handled by smart grids technologies.

1.1.4 The concept of smart grid

The incredible development of Information and Communication Technology (ICT) infrastructure in the last decades represents a strong momentum for grids as well. For example, fiber optic cables (the backbone of the communications network) have grown from 95 billion km in 2000 to 180 billion km in 2011 worldwide. The growth of global internet users grew from 0.413 billion (5.8 % of the world population) in 2000 to more than 4.21 billion (55.1 % of the world population) in 2018 [32]. The mobile connection has increased almost nine times to 6.3 billion in 2012 from the 2000s. The development

of ICT infrastructures and sensor grids constitutes an Active Management Network (AMN) which is the foundation of smart grids technologies.

In Vietnam, according to [43], in 2016, 143 million people have a mobile connection, and internet users represent 64 % of the population in 2018. The average download/ upload speed is 22.2 Mbps and 6.72 Mbps respectively (compared to the average upload speed in the world which was 9.1 Mbps) in 2018.

Smart grid refers to the evolution of the power grid in which the information and communication technology is integrated and participates in the control and management of the power system, as represented in Figure 1.3. The primary goal is to balance electricity supply and demand at all time and to provide a secure, sustainable and competitive electricity supply for consumers.

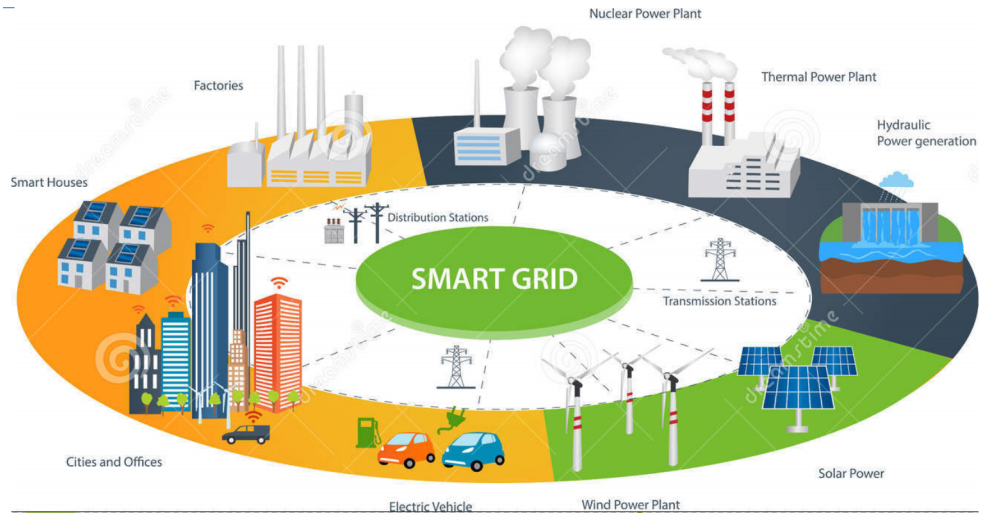


Figure 1.3: The concept of smart grid [60].

There are a lot of components in the power grid which could be controllable objects such as renewable energy sources, factories, smart houses, smart buildings, electric vehicles, etc. All of the components participate in the decision of electricity usage in the system. The coordination is made possible by using smart grids technologies.

In order to characterize the smart grid, a framework is proposed in Figure 1.4, allowing to change the traditional power grid from passive to active [36]. It comprises three main layers: application, power and ICT.

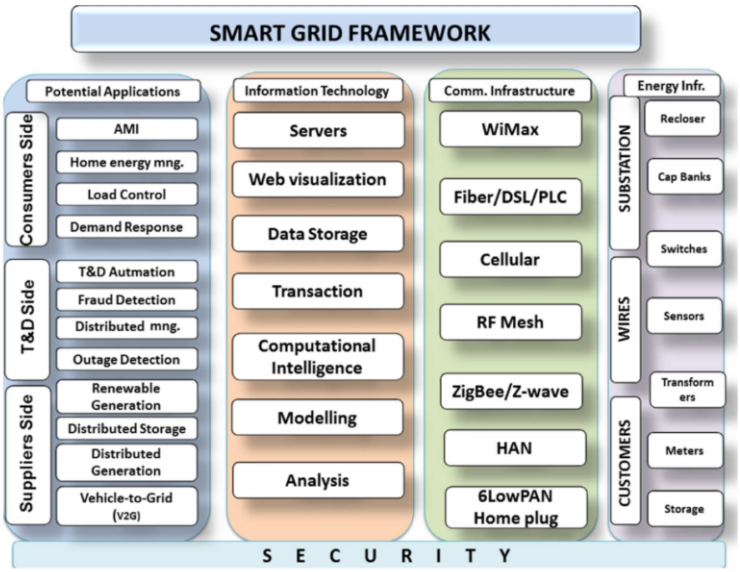


Figure 1.4: smart grid framework [36].

1.1.4.1 Application layer

It includes advance applications providing interoperability and an easier operation of the system. It can be mainly divided into five major categories, presented in Figure 1.5 [61], whose positive impact on the grids are described below.

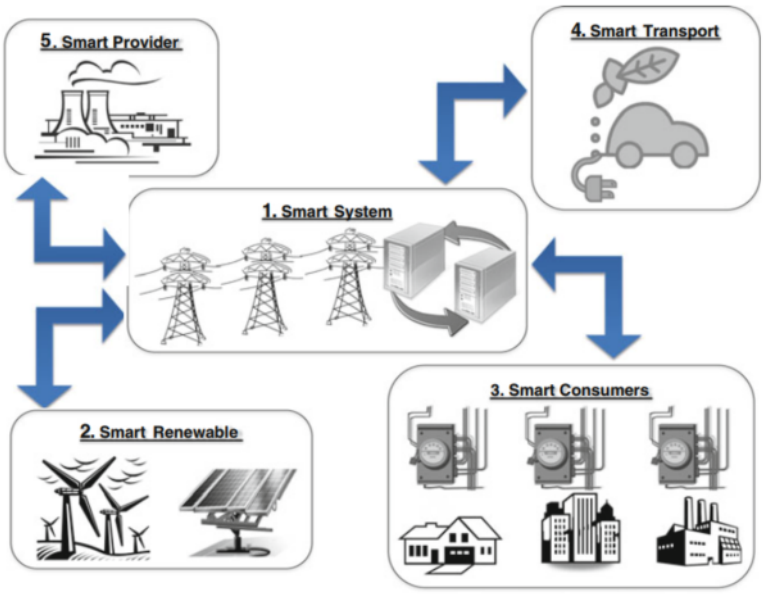


Figure 1.5: Major applications in smart grid [61].

Systems: Improved resilience to faults such as disruption, attacks, and natural disasters.

Renewables: Improved accommodation and management of various power generations. Limit of integration of renewable energy sources into the power system is enhanced while ensuring the stability of the system.

Customers: No longer passive, they can be informed about their consumption, and adapt to dynamic pricing. They could participate in demand-side management.

Transport: Capacity to manage electric vehicle charging in order to reduce peak load.

Service providers: A large collected data-set could be used to evaluate the optimal compromise between economic criteria and customer needs.

1.1.4.2 Power layer

Power generation, electrical devices, and substations in energy transmission and distribution systems are all contained in the power layer. That part is considered as the object of control, where information is collected to serve for decisions in the application layer. According to the US National Energy Technology Laboratory (NETL), the deployment of a smart grid needs a significant quantity of technologies such as [66]:

Sensing and measurement: Includes vital technologies such as Phasor Measurement Unit (PMU), Smart Meter (SM), and Advanced Metering Infrastructure (AMI) to acquire data for monitoring, protection, and control. However, they are quite expensive to meet the requirement for full coverage monitoring in wide areas.

Advanced components: Electronic devices are used in the power system to stabilize the system facing faulty events [40].

1.1.4.3 Communication layer

The communication layer is presented as the heart of the smart grids technology. It provides interconnections between all devices and systems. That helps data ignition, and potentially increases the reliability of the system by collecting data in real-time and acting on it (while increasing the sensibility to cyber-attacks).

This communication layer consists of a backbone, a back haul, and access points, as presented in Figure 1.6 [26].

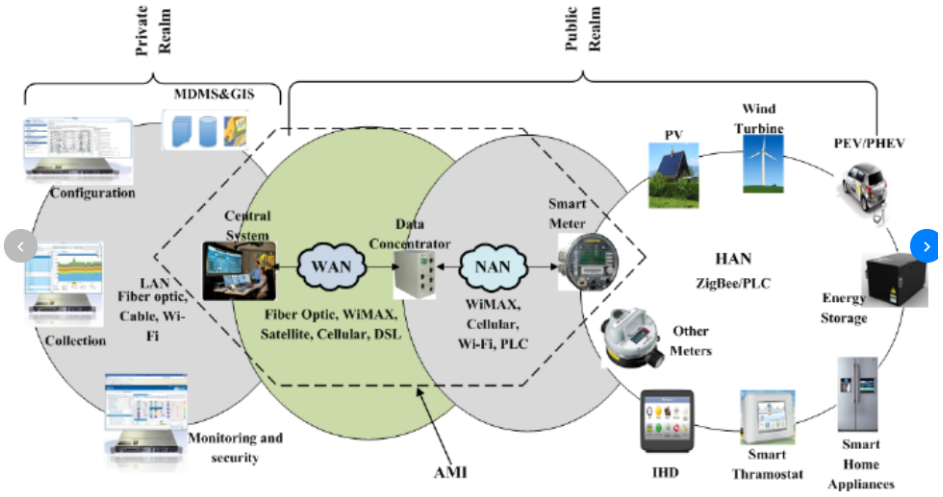


Figure 1.6: Categories in the communication layer of smart grids [26].

Wide area network (WAN): It is considered as a backbone of the system, providing communication between the electrical utility and the substations. It requires a high bandwidth communication grid to deal with long distance data transmission and fast response time of some applications such as substation automation. Fiber and microwave communications are preferred for this grid.

Neighbor area network (NAN): It acts as a bridge between the premises of customers and substations with collectors, access points, and data concentrators. IEC 61850 is providing the interoperability between intelligent devices [79]. Fiber optic or WiMAX can be used in this grid [12].

Home area network (HAN): It supports home electrical appliances and smart meters. Low-bandwidth, low latency, and flexible connections are preferred for HAN such as ZigBee, WiFi, HomePlug, GPRS, 3G, 4G, etc [1].

The transmission of real-time measurement data causes a big challenge of bandwidth use as well as investment cost for communication infrastructure in the future [10].

1.1.5 The concept of microgrid

To manage the distributed renewable energy resources as well as to improve the resilience and autonomous operation of the power system, a big electrical grid could be divided into smaller parts, called microgrids. Also, where no grid is available, microgrids are generally

the most economic (and only) option. Thus, microgrids are sometimes considered as the building blocks of smart grids [38]. The microgrid can be integrated and operated in a low (240 to 600 V) or medium (2,4 to 69 kV) voltage and its principle is presented in Figure 1.7.

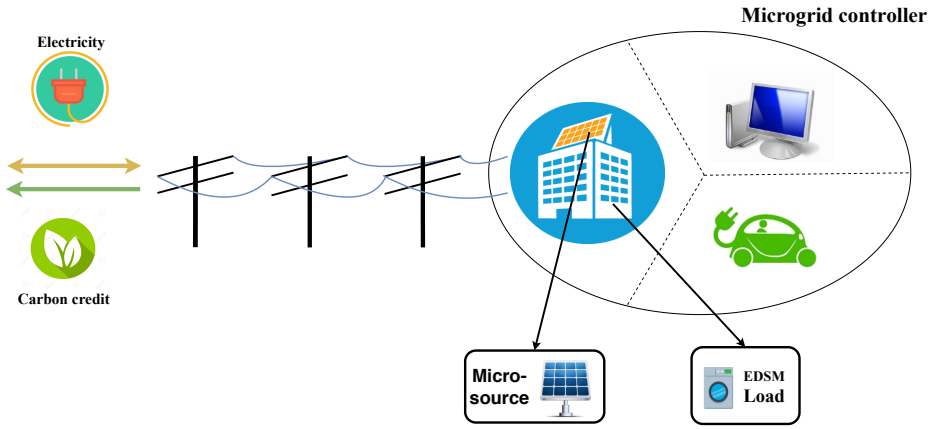


Figure 1.7: A sample microgrid architecture.

- A microgrid is a supply-side platform in a distributed system.
- It can contain distributed energy resources, storage devices, and loads.
- It can be interconnected to the main grid (grid-connected mode), but in case of a fault, it can entirely operate in an autonomous way (islanded mode).
- There are also "smart" functionalities to support, monitor, manage and control this system, called the microgrid central controller (MGCC).

The concept of multi-microgrids is extended from the microgrid concept in a high-level infrastructure shown in Figure 1.8. It consists of several low voltage (LV) microgrids and distributed generation units which are connected to adjacent medium voltage (MV) feeders. In this context, each MGCC must be operated in a coordinated way to achieve the optimal control and management in the overall system. The central autonomous management controller (CAMC) can be installed at the MV bus, or at the high voltage level of the substation. It is responsible for managing, for the distribution system operator (DSO e.g. ENEDIS in France), the multi-micro grid system (M-MGCC). Meanwhile, the adaptation of the energy distribution management system (EDMS) is in charge of the whole distribution grid.

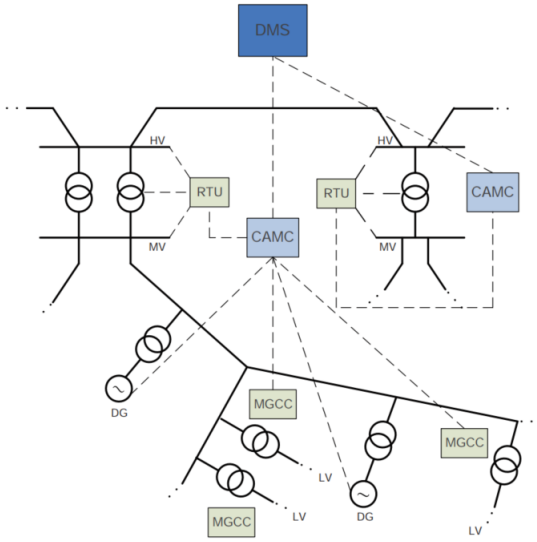


Figure 1.8: The hierarchical management structure of multi-microgrids [38].

The hierarchical management structure is proposed in the multi-microgrids concept because it supports a flexible control and management architecture and a large amount of data processing. Whereby, a central controller could collect data from multiple devices by delegating this task to lower level controllers. Moreover, in case the small system is cut-off from the main grid, the CAMC will be in charge of the role of the DMS. A transition has to be properly operated in that case.

1.1.6 Controlling and monitoring the power grid in real-time

In future smart grids, if the main grid is divided into multi-microgrids, the stability of each microgrid will be a fundamental condition for the stability of the main power grid. The real-time response of the microgrids should be controlled as well as to deal with all variations of consumption or production.

The functionalities of the microgrids can be classified in three levels, from individual control to global control in a normal operation, presented in Figure 1.9 [38].

Primary control: It contains all the local functionalities, performed in a single generator, storage device, or controllable load. The aim is to keep the local balance between demand and supply. The methods used in primary control can be primary voltage/frequency control, primary active/reactive power control, droop control, battery management, or protection functions [29].

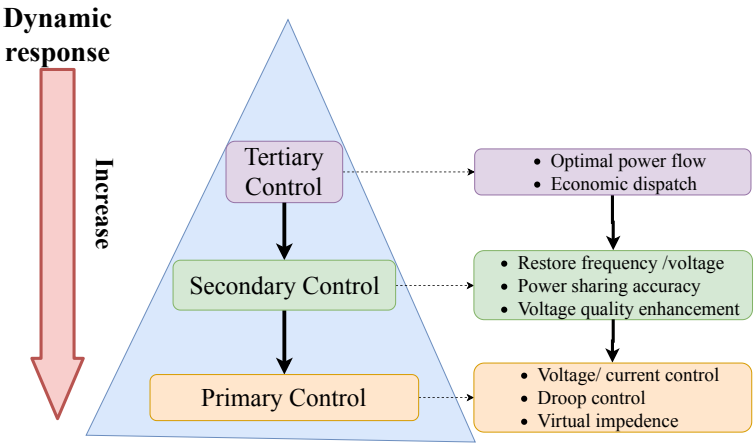


Figure 1.9: Three level control in a microgrid [38, 37].

Secondary control: It contains functionalities within a microgrid to improve the electrical quality and security of the system. These functionalities can be load and renewable energy resources (RES) forecast, load shedding/management, unit commitment/dispatch, voltage/frequency control, active/reactive power control, security monitoring, black start, etc.

Tertiary control: It concerns the global system in which the energy market is taken into account. This level decides the import or export of energy for each microgrid in the multi-microgrids system on a long-term perspective.

The primary control is performed on a local machine; meanwhile, the secondary and tertiary control need communication with a central (or partially decentralized) controller to take a decision. The longer the computation time is, the slower the response time of each level is. The dynamic response time decreases from the first level to the third level.

1.1.7 Data management in smart grids

Supervisory control and data acquisition (SCADA) plays a critical role in the power grid, particularly in smart grids. It allows observing, forecasting, detecting, analyzing, and preventing faults in large power systems. Thus, the oversight of SCADA helps power systems to improve their overall efficiency. However, the increment of the numerous smart devices integrated into the grid to measure and collect data causes many problems. It is impossible for a SCADA to handle all transient oscillations of the power system over a too short period of time [90]. Moreover, data coming from the enormous number

of sensors lead to the problem of data management in smart grids. Without talking about cyber-security, the technical challenges are:

Volume: For example, at the end of 2015, the total phasor measurement unit (PMU) installation reached more than 1380, covering nearly 100% of the US transmission system. In China, this number reached over 5000 PMUs by the end of 2018 [72]. There are hundreds of millions of smart meters (SM) installed globally. According to Navigant Research [61], there were 139 million of SM installed in 2012, and this number will increase up to 377 million in 2020. In Vietnam, 65 % of the houses are equipped with SM using PLC and GPRS/3G [1]. Moreover, the automatic meter reading (AMR) has been deployed to replace the traditional twice-a-year-reading meter. Each meter reading more than four times an hour means 96 more data per day, and more than 2880 times more data in a month just considering basic metering. In reality, the rapid increase of AMR, PMU, SM and other advanced measurement devices such as intelligent electronic devices (IDE) or digital fault recorder (DFR), produces an enormous volume of data in power systems to store, analyze, and visualize. As an illustration, 100 PMUs with 10 measurements generate over 50 GB of data per day at 60 Hz [46].

Variety: Data analysis in smart grids deals with multi-sources data-sets such as energy resources, geographic information, demand response management, customer information, etc. They have various formats, modalities, and representation.

Velocity: Data collected from a huge amount of SM is done with a short time sample. In particular, in some urgent applications such as fault, transient oscillation detection, restoration and fast voltage/frequency control, the time scale of the reaction should be in less than milliseconds. That requires lots of data transmission in the communication network, which can affect the latency and data loss. Some kind of prioritization could be put in place to cope with this.

1.2 Control infrastructures for microgrids and smart grids

1.2.1 Control and management strategy

Smart grids indirectly consist of a complex and considerable number of interconnected distributed generators (e.g. distributed energy resources (DER), storage system, fuel generators) and subsystems. Three strategies, presented in Figure 1.10, are usually proposed to control such systems: centralized, distributed and decentralized [37].

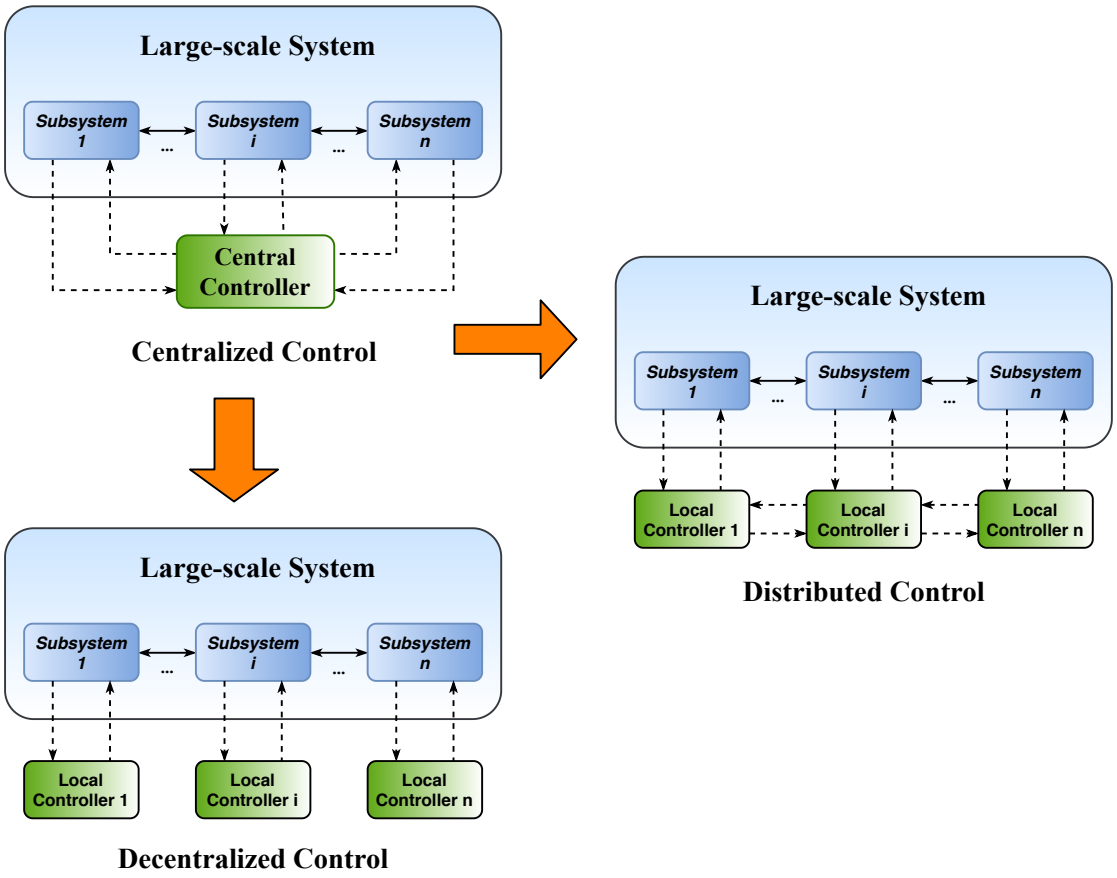


Figure 1.10: Control and management strategy for smart grids [37].

Applying the centralized control strategy (like for traditional grids) to smart grids could be ineffective and difficult to deploy. The limits are the communication ability among the subsystems as well as the computation ability in a single controller/node. In addition, a failure at the central node will make the centralized system unreliable at a very large scale. The main issue is how to deal with the scalability of the number of sensors and the generated data which increases at an exponential rate [3].

The decentralized control strategy is available for each subsystem. Accordingly, each controller is designed and operates independently without a mandatory interaction with other subsystems. Decisions are made based on local available information. However, this control still has some drawbacks, such as the geographical limitation of information, reducing the control performances. Moreover, the local control focuses on the operation of its subsystem. This could affect the global stability. An example of that situation is the widespread blackout of 2003 in north-east America [33]. On August 14, 2003, an accident occurred in the high voltage line which supplies electricity to the Ohio area, causing the failure of the entire line. Because of a software bug, this accident was not alerted to operators. The electrical demand in Ohio supplied by the line was transferred to other lines in order to maintain the power balance in this area, overloading their protection and in the end causing a cascading failure.

In that context, the distributed control strategy presents an interesting compromise. Based on the development of an advanced information grid, it allows communicating data with other subsystems, which can have essential global knowledge while functioning also locally. A distributed control strategy could be considered as the combination of the advantages of a centralized control and a decentralized one.

1.2.2 Distributing elements of the infrastructure of grids

1.2.2.1 Distributed systems

Smart grid technologies are naturally compatible with distributed system, i.e. a set of distributed components located in the wide-area grid and cooperating in managing the power grid. Each component, called a node, runs as a computing unit. The operation of the node depends on the operation of other nodes in the grid. The major goals of a distributed system are [88]:

Making resources accessible: The distributed system allows nodes to access remote resources or efficiently share their own resources. That makes use of the available components in the system easier. Besides, these nodes cooperate and exchange information over the grid. That potentially enhance the overall reliability of the system.

Transparency: A node can estimate the image of other nodes without concealing the details of their location, access, migration, concurrency, failure, relocation, persistence and resources.

Independent failure: Failure of a single node does not affect the whole distributed system by default.

Openness: The system can be flexible, making it easier to (re)configure and add new components without changing the existing infrastructure.

Scalability: Distributed systems aim at being scalable regarding geography, administration or size.

The smart grid scenario development towards a distributed system is essential to deal with the scalability resilience, and complexity of the energy system.

1.2.2.2 Multi-agent systems

The multi-agent system is a method proposed to manage distributed systems. It is a set of agents that can communicate with each other. According to [74], there are four types of agents, whose relation is presented in Figure 1.11: the control agent, the DER agent, the user agent and the database agent. Open-source platforms such as JADE, Zeus, SPRING Tracy, and Aglets [42] are supporting various languages dedicated to the communication between agents. The main characteristics of multi-agent systems are:

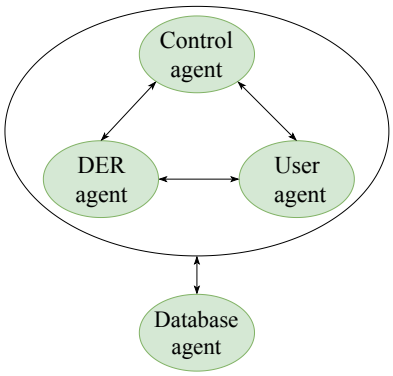


Figure 1.11: Classifying agents in a multi-agent system.

Autonomous agents: They perform a partial control of their actions and internal state without the intervention of humans or external devices.

Social: Each agent can communicate with other agents.

Local view: No agent has a complete global view.

Decentralization: No agent is designated as the central controller. Computation is asynchronous. Each agent can execute its tasks independently, without waiting for a central control signal.

The multi-agent system supplies a method to communicate among agents. That allows connecting independent agents into a unique entity to have a global view. However, the deployment of centralized algorithms over this system is a challenge. One of the proposed methods in a smart grids context is using distributed algorithms.

1.2.2.3 Distributed algorithms

Many researches apply distributed algorithms to support control strategies for power systems [82, 57]. First of all, consensus algorithms should be taken into account. They are used to achieve agreement on a single data value among distributed processes or systems. There are various consensus algorithms presented in the context of smart grids which can be non-iterative or iterative algorithms.

Simple consensus [50], K-set[50], Paxos[50], Raft[68] are proposed for non-iterative consensus algorithms. They help increase the scalability of the system and reduce the risk of single-point failure by dividing the system into groups and electing virtual leaders of each group. However, it's hard to deploy these algorithms because of the synchronization among different groups and data failure detection.

For iterative consensus algorithm, the consensus problem is stated in a grid $G = (N, E)$ with N representing a set of nodes and E representing a set of edges, $N_{ij|i,j\in E}$. Each node has an initial real scalar $x_i(0)$. The average value of x is determined at the $(k+1)^{th}$ iteration.

$$x(k+1) = W \times x(k) \tag{1.1}$$

with W the weight matrix, and $x(k)$ the state vector at the k^{th} iteration.

The consensus algorithm converges when the state of all nodes at the current iteration is equal to their state at the previous iteration. The convergence of the consensus

algorithm depends mainly on the weight matrix W and the grid configuration. There exist popular heuristic weight matrices assuring that the consensus algorithm is always converging, for example the *constant edge weights*, the *maximum degree weights* and the *metropolis weights* [99].

For more complex problems like the ones encountered in the context of smart grids (for example, optimal power flow), there exist also dedicated approaches for the distributed algorithm. For instance, the Gauss-Seidel (GS) and the Gauss-Seidel Relative Error (GS-RE) [7], the Diagonal Quadratic Approximation (DQA) [84] and the Alternating Direction Method of Multipliers (ADMM) [15]. The ADMM is suggested as the adequate algorithm to approximate the minimization of the augmented Lagrangian by a decomposition approach [24]. ADMM with Proximal Message Passing (PMM) [65] allows decoupling the optimization function into sub-problems in order to reduce the number of local variables and the complexity of the problem. In each small problem, an algorithmic loop will optimize its objective function with the local variable while non-local variables remains constant (converged).

To conclude, there is a good variety of distributed algorithms applied in control and management problems related to smart grids. However, this approach has still some drawbacks, especially regarding optimization problems. There are many centralized algorithms but less distributed algorithms for power management applications. Besides, distributed algorithms have a slow convergence rate when the system scales up. In addition, they usually lack the programming abstraction to be deployed in practice.

1.2.2.4 Distributed databases

Database management The management of data in active grids is operated mainly through a database management system (DBMS) or sometimes simply a file system [20]. In both case, the basic functionalities of data management are:

Scheme creation: Defining data format and relationships between data.

Data modification: Inserting, updating or deleting data in database.

Query and access: Retrieval of stored data as requirement.

Backup and recovery: Preventing data loses and improving data security.

Performance optimization: Making the retrieval process faster by using specific data structure and algorithms.

For the DBMS, relational database is the most common standard. Some commonly used relational DBMS are Microsoft SQL server, Oracle, SAP base, PostgreSQL, MySQL [92]. Also, structure query language (SQL) is a common interface to retrieve data from relational DBMS.

Nowadays, a post-relational database system call NoSQL becomes more and more common. The NoSQL database systems includes document oriented databases (e.g. MongoDB), XML databases (e.g, BaseX), key-value stores (e.g, Apache Cassandra), object-oriented databases (e.g, db4o), and column-oriented databases (e.g, c-store) [63]. The NoSQL database system uses conventional programming languages like C#, C++ or Java. In addition to the NoSQL database system, parallel and distributed file systems such as Apache Hadoop or Google MapReduce are getting popular [76].

Distributed database management The components of smart grids (renewable resources, substation transformers, data concentrators, and smart meters) are by essence geographically scattered. Moreover, a significant quantity of sensors participates in the stable operation of the power grid. This generates a massive volume of data to support the decisions for control and management of the grid. But that also poses a big issue in data management. How to process and analyze the massive and diverse data? As the size of the data started increasing, maintaining this data in central data stores became difficult. In this context, a distributed database architecture can show its advantage in time responses of local management in comparison with a central database. Besides, a distributed data processing infrastructure could increase the reliability and efficiency of smart grids [100, 102, 45]. This will be more and more true with the increasing volume of data.

Distributed database management allows managing various cooperating databases. It provides a structure to share and access data via a common interface. It transparently deals with data independence, replication, and fragmentation. As an example, an interesting architecture of a distributed database management was presented in [54] that helps smart grids being more resilient to attacks. It also reduces the ICT network usage

and the streamline query processing. Moreover, distributed real-time database management is supposed to better attain stability and accuracy, as shown in many existing systems, like for example in China [55].

Distributed relational database design The relational database is a database where the structure is in the form of tables. A table is declared with names and attributes, called *relation schemes*. A relation R is defined from an attribute set $A_1, A_2, ..., A_n$ to a data-item set $< d_1, d_2, ..., d_m >$. Considering for example the data model of a smart meter, the entities to be modeled are the measured parameters of the smart meter. It includes the identifier (ID), the voltage parameter (Volt), the current parameter (Cur), the power parameter (Pow), the power factor (PFA). This relation scheme can be defined as presented in Figure 1.12.

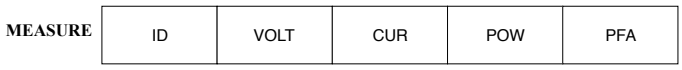


Figure 1.12: Sample database scheme for a smart meter.

In the relation scheme `Measure(ID, Volt, Cur, Pow, PFA)`, there are five attributes: `ID, Volt, Cur, Pow`, and `PFA`. The values of a row with all schemes is called a tuple and corresponds to $< d_1, d_2, ..., d_5 >$. The *key* of a relational scheme is a non-empty subset of an attribute so that the value linked to this attribute is unique in the whole domain. For example, in above example, `ID` is a key, and the empty table structure corresponds to the relation scheme. When the table is filled with rows, it corresponds to a relation instance. A simple illustration is proposed in Table 1.4. There main relational algebra operators are presented below.

Table 1.4: Illustration of a sample relation scheme instance for a smart meter.

ID	Volt	Cur	Pow	PFA
0001	220	10	1000	0.99
0002	221	12	1500	098
0003	219.5	5	950	0.985
0004	220.5	15	1680	0.995

Selection: It produces a horizontal subset of a given relation. The subset contains all tuples that satisfy a condition. The selection from a relation R is: $\sigma_F(R)$, where F is a condition, and R is a relation. Considering the relation scheme instance

`Measure` shown in Table 1.4, an example of the selected tuples is presented in Figure 1.13.

$\sigma_{ID=0001}(Measure)$

ID	Volt	Cur	Pow	PFA
0001	220	10	1000	0.99

Figure 1.13: Example of Selection.

Projection: It produces a vertical subset of a relation. That is a column selection over which the projection is performed. The projection of the relation R over the attributes (A, B) is denoted as $\Pi_{A,B}(R)$. Note that the result of a projection might contain tuples that are identical. For example, the projection of the relation `Measure` shown in Table 1.4 over attributes `ID` and `Volt` is depicted in Figure 1.14.

$\Pi_{ID,VOLT}(Measure)$

ID	Volt
0001	220
0002	221
0003	219.5
0004	220.5

Figure 1.14: Example of Projection.

Union: For two relations R and S, it is denoted $R \cup S$, and is the set of all tuples of both relations. As in the case of a projection, R and S have the same scheme.

Set difference For two relations R and S, it is denoted $R - S$, and is the set of all tuples that are in R but not in S.

Cartesian product: For two relations R with k_1 tuples and S with k_2 tuples, it is the set of $(k_1 \times k_2)$-tuples, where each tuple is a concatenation of one tuple of R with one tuple of S. The Cartesian product of R and S is denoted as $R \times S$.

Intersection: of two relations R and S $(R \cap S)$ consists of the set of all tuples that are in both R and S: $R \cap S = R - (R - S) = S - (S - R)$.

Join: For two relations; it is a selection formula over the Cartesian product of the two operand relations: $R \bowtie_F S = \sigma_F(R \times S)$.

In the distributed relational database's design, a relation can be distributed in the system with fundamental fragmentations which are presented as:

Data fragmentation There are two fundamental fragmentation strategies: horizontal and vertical, as depicted in Figure 1.15. However, there is a possibility of nesting fragments in a hybrid way.

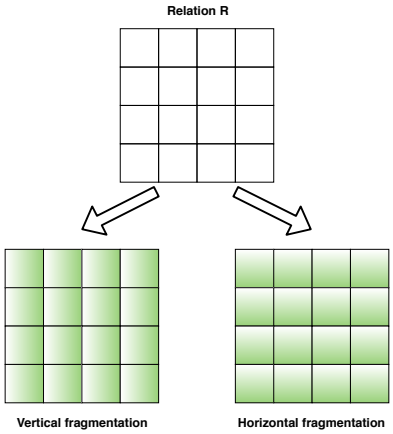


Figure 1.15: Data fragmentation.

Horizontal fragmentation: For a relation R, it produces the fragments R_i, each being the result of selector in R with a specific condition C ($\sigma_C R$) and usually a subset of the tuple of the relation. The reconstruction of the relation R through horizontal fragments gives $R = \cup R_i, \forall R_i \in R_1, R_2, ..., R_w$.

Vertical fragmentation: For a relation R, it produces the fragments R_i, each containing a subset of R's attributes as well as the key of R. The reconstruction of the relation R through vertical fragments gives: $R = R_1 \bowtie R_2 \bowtie ... \bowtie R_w$.

For a hybrid fragmentation, we have $R = (R_{11} \bowtie R_{1i}) \cup (R_{21} \bowtie R_{2j}) \cup ... \cup (R_{w1} \bowtie R_{wk})$.

The declarative paradigm is used for data manipulation in distributed DMS in smart grids, in which the database is active and reacts immediately to any change in data. Besides, declarative programming has a high-level programming abstraction, supports distributed programming in distributed DBM. The logic programming language is one of the categories of the declarative paradigm.

1.2.2.5 Logic programming language

Most of the proposed algorithms for traditional and smart grids rely on imperative paradigms which are generally programmed in common languages (e.g. Java, Python or C). The calculations and communications in this paradigm are carried out sequentially thanks to the time step rather than the actual meaning of data [67]. The addressed problem is not perspicuous because the algorithm states the order in which the operation occurs (i.e. "how to do") and not how to react to information (i.e. "what to do").

Nevertheless, elements in smart grid are almost eventual. This paradigm leads to computations and communication redundancies. In that case, an algorithm able to react to a concrete problem is more efficient than *sequential actions* to manage the system. Thus, in this context, an algorithm supporting the declarative paradigm should be more convenient than those supporting the imperative one.

A logic program is a part of the declarative language performed based on formal logic. It is a set of logical forms that consist of rules. The main examples of logic programming languages are Datalog (one of the first logic programming language used as query language for deductive databases) [31], NDlog [56] and Netlog [2]. These languages are written in the form of *horn clauses*: ``Head :- Body''.

Where `body` is a set of terms or a set of sub-goals B_1, B_2,...,B_n. The head part will be deducted if all the sub-goals are approved.

A `fact` in logic programming is a tuple of data items, and declared as a group of attributes and their data type. For example, coming back to the smart meter example of Table 1.4, a declaration of `fact` named `Measure` will have three properties: identify (with datatype integer), voltage (with datatype float) and frequency (with datatype float), expressed in Listing 1.1.

```
Measure(ID: int, Voltage: float, Frequency: float).
```

Listing 1.1: A sample "fact" declaration.

A `fact` present in a rule with variables or constants is called an `Atom`. An `atom` with all variables represents the whole data items of the attributes in a `fact`, as expressed in Listing 1.2. An `atom` with known values represents the data tuples in the `data_type` where the corresponding attributes are equal to these values.

```
Measure(i, v, f).
```

Listing 1.2: A sample "atom".

An algorithm of logic program works based on the Kowalski's principle [47] : ''`Program = logic + control`'', where `logic` is a set of rules which defines the behavior of the system as setting in `data_type`, while `control` determines the order of the execution of the rules which affects the efficiency of the program. The main advantages of the logic programming language are:

- High-level declarative language, close to a natural reasoning;
- Reactive to changes in data;
- Recursive;
- Can be easily deployed into distributed processing programs.

Logic programming languages have received a great deal of attention in research and development, but not in Electrical Engineering, which is the main argument motivating the present research.

As an illustration, to express: "when there is a modification in the measured data at the i^{th} node, then update the value to its neighbor j, and save it in `NeighborMeasure`" in Datalog [31], the rule is expressed in Listing 1.3.

```
NeighborMeasure(j, i, v, f): Measure(i, v, f), Neighbor(i, j).
```

Listing 1.3: Illustration of implementation with Datalog.

The scheme of the rule is presented in Table 1.5.

Table 1.5: Example of a scheme of a rule in Datalog.

Atom	**scheme's description**
Neighbor(i, j)	Neighbor(Node, Neighbor)
NeighborMeasure(j, i, v, f)	NeighborMeasure(Node, Neighbor, Voltage, Frequency)

Several requirements exist in declarative languages. For example for Datalog: atoms can not hold complex terms as arguments of predicate (for example `A(f(1), 2)` is not

allowed); every variable appearing in the head part must appear at least once in a non-arithmetic positive atom in the body of the rule; and every variable, before appearing in a negative atom, must appear in a positive atom. Both Prolog and Datalog are implemented in a local database but Datalog expresses simple relations between data rather than the computation support that Prolog can provide. However, there is no option in both logic programming languages for multi-databases. This limitation shows that these languages are not completely suited with the needs of this work, aiming at developing a language dedicated to smart grids.

More recently, Netlog [34] and NDlog [56] have been proposed an extension of Datalog supporting network protocols and P2P applications. Netlog defines explicit primitives for communication, storage, as well as location instruction and destination specifier. Netlog is based on a semi-naive evaluation, which triggers only rules over related inputs. The affection operator in front of the head part determines where the result is affected. The character "↑" is used to send the result of the rule to another location, the character "↓" is used to store the result in the local database and the character "↕" do both actions. For example, to express the same rule as Listing 1.3 with Netlog rather than with Datalog, assuming that the computation takes place in node i and the result of the head part is stored in node j, we obtain Listing 1.4.

```
↑ NeighborMeasure(@j, i, v, f):- Measure(i, v, f), Neigbor(i, j).
```

Listing 1.4: Illustration of an implementation with Netlog.

The "@" operator added before a variable in the body part or in the head part is a *destination* specifier, which indicates where the result will be stored.

Netlog supports arithmetic operations with aggregation, assignment and non-deterministic constructions such as negation, random choice, and consumption. For simplifying the executions of the rules, they are grouped in modules. However, the optimization of the data storage and the rule control in Netlog has to be improved to accommodate the smart grid's applications.

1.2.2.6 Rule-based system

A rule-based system (RBS) is a system that uses rules as the representation of knowledge coded into the system [9]. A RBS consists of a knowledge base and an inference engine, as shown in Figure 1.16. The knowledge base, described in a declarative programming language, contains rules and facts. Each rule indicates a specified behavior of the system when particular problem-solving data are entered. Each rule is a form of *condition-action*. The body part of the rule consists of a conjunction of condition elements. The head part specifies the action that will be performed if the conjunction in the body part is true.

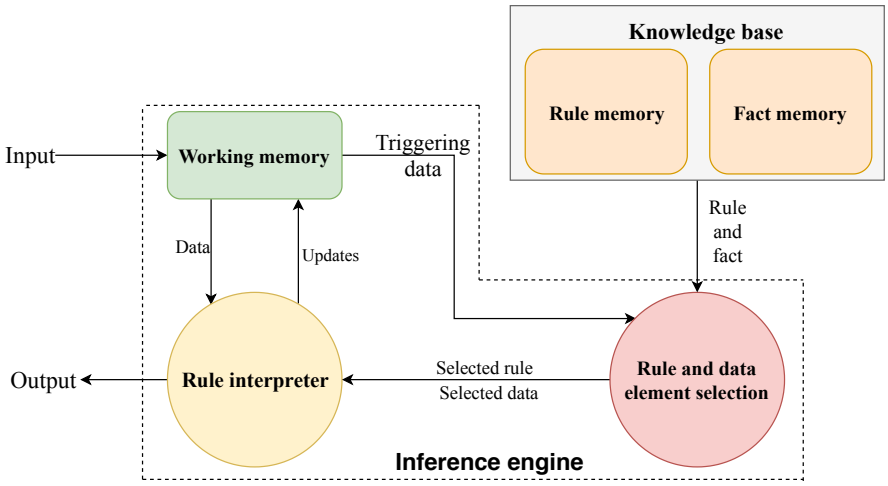


Figure 1.16: The basic features of a rule-based system.

The basic cycle of inference engine contains two main phases: selection and execution. The selection phase chooses all rules firstly in a knowledge base satisfying the content of the working memory and groups them into a conflict set. Then, based on the priority and specificity of each rule, the system will select a rule in the conflict set for execution in the next phase. In the execution phase, the system interprets the selected rules into action or dynamic fact which is stored temporarily in the working memory. This cycle will be repeated until no more rules are executed or the goal is achieved [39].

System storage in a RBS includes long-term static data and short-term dynamic data. The long-term data storage is a knowledge base which consists of rules and facts. The input data will trigger conditions for rules and data selection. These conditions present the logical expression and the proposition or data in terms of relations. Most of functionalities in RBS are dedicated to distribute their logic over numerous independent

condition-action rules, monitoring dynamic results for triggering patterns of data, determining their sequential behaviors by selecting their next activity from a set of candidates-triggered rules, and storing their intermediate results exclusively in the global working memory and/or in the fact memory.

The fundamental function of a RBS is to produce an output. Whatever the considered case is, the RBS is employed to determine its overall activity and built-in control procedure to provide the order to execute actions. The RBS addresses some shortcomings in the programming technology, that is:

- Non-specificity of the programs;
- Changes in the principles of operation;
- Lack of expert participation in exploiting the computer program.

1.2.2.7 Distributed rule-based system

As aforementioned, a distributed computing language is preferred in the context of distributed control strategies for smart grids. The logic programming language is one of the first languages proposed for distributed computing because of its simplicity. From a construction perspective, rule-based distributed systems such as OPS5 [17], MAGSY [27] are set in the context of an individual agent architecture and respect to multi-agent systems [9]. Each node is a rule-based system, and the coordination of the nodes can be achieved either via shared memory or via asynchronous message passing. The distributed system requires an algorithm which allows realizing computations over the whole system. Lastly, it is interesting to mention the RETE algorithm [48], presented in Figure 1.17. This algorithm appeared as a fundamental theory for developing parallel/distributed processing in big data applications. The RETE algorithm separates data into objects and performs the relation of data (e.g. rules) by graphs, which construct a *RETE network*. During the run time, the changed data in an object will be verified based on its graphs in the RETE network. The satisfied rules will then be ready for execution. This process is called *Evaluation*. In the *Execution* state, a set of satisfied rules will be fired one by one with respect to their priority.

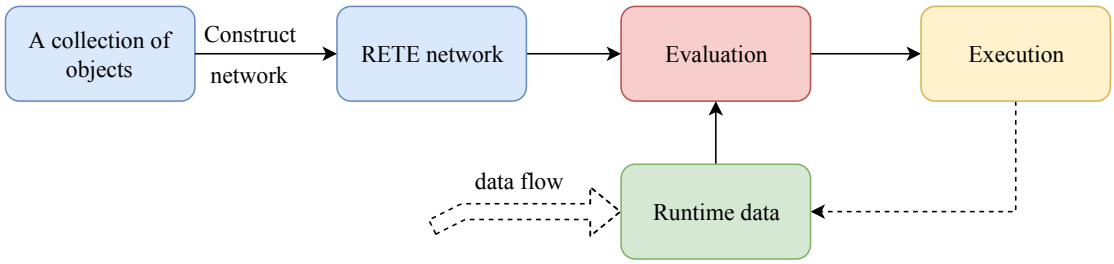


Figure 1.17: Principle of the RETE algorithm.

The LEAPs [11] and Treat algorithms [64] both improved the RETE algorithm by dealing with rules activation in each rule cycle. Besides, the Parallel Prolog program [34, 80] presents data-parallel approaches supporting concurrent computations in a database environment. But these works are conducted in one simple machine that cannot deal with the issues of volume when the number of facts becomes too large. The Lana-Match algorithm [8] enhanced the performances of the Rete-Match algorithm by running it in a distributed memory system with the Controller-Slaver paradigm. However, it lacks flexibility and causes possibly a bottleneck phenomenon and memory intensive processor needs in the controller. Delta-Prolog [19] developed a distributed prolog language in a distributed memory set, but the subprograms are still distributed manually. The Rule Matching algorithm [101, 96] proposed a distributed processing in the Spark platform to deal with big data. Though these algorithms allow sharing the computational burden in rule-based systems, they do not supply the description for data allocation as well as any support for directed communication. As a summary, a solution dedicated to smart grids still lacks, which motivates the present thesis.

1.3 Conclusion

Smart grids development is a prerequisite for the traditional power grid to better integrate renewable energy sources. However, this development demands detailed and innovative researches to accommodate the evolutions. The centralized mechanism of the traditional grid shows a lot of shortcomings about the problem of big data processing. Meanwhile, some results in dispersal grid management present potentials of scalability. So, the critical question is how to transform the system management mechanism from centralized to distributed?

In order to develop this strategy, in addition to the actual availability of the infrastructures (which is a hardware problem), the critical question is how to integrate traditional grid controls and managements into this paradigm (which is a software problem)? Distributed algorithms allow deploying centralized algorithms over distributed data, but they still have limitations. They have for instance a slow convergence and a lack of programming abstraction, all of which make them less practical.

The main objective of this thesis is to aim at proposing a declarative and reactive language and a programming methodology that could contribute to the evolution of current power grids towards a smart distributed system. This approach promises to overcome drawbacks of the current researches in a distributed implementation of smart grids while conserving centralized expression of current regulations. The detail of the development is presented in the next chapters.

Chapter 2

Smartlog: A declarative language for smart grids data management

2.1 Introduction

Smart devices associated with various components of the power system, such as small generators, renewable energy resources, storage devices, and so on, are a critical part of active management network (AMN). The AMN participates in most controls in the distribution grid, such as active and reactive power controls, voltage management, and energy balancing [30]. Each node in the network can collect sensor data and communicate with grid operators. AMN development aims at optimizing the operation of the power system and increase its resilience.

However, the management of grid data in this system poses a challenge, especially developing countries like Vietnam. The issue is to manage the volume, the variety as well as the velocity of data in the limitation of current infrastructures. To meet these challenges, significant investments in computing power and communication systems are required. That is why the centralized data management is no longer sufficient, especially for real-time applications. A more efficient solution would be a distributed data management. Collected sensor data are stored in distributed locations, near sensors, to reduce the data transfer costs, to increase the data velocity, and enhance the reliability and accuracy of the overall data system.

In this context, each node of the smart grid should play the role of an elementary cell. Based on that, we propose a new high-level programming language called *Smartlog*. *Smartlog* is a declarative rule-based distributed data manipulation language (RBDML). It is developed based on the Netlog language [34], itself based on Datalog [31]. It is improved to accommodate the needs of smart grids. Some major properties of this language such as compactness, simplicity, and scalability are highlighted in this thesis. They should be adaptable to many power systems configurations (defined using topology, characteristics of loads and sources, etc.) and aiming at efficiently operate the power system in real-time with limited bandwidth.

The chapter is organized as follows. In Section 2.2, we present the architecture of each node and of the network in which *Smartlog* can operate. In Section 2.3, we define the procedural syntax and semantics of *Smartlog*, and explain how *Smartlog* can support a distributed programming mechanism. Ultimately, in Section 2.4 and Section 2.5, we describe the *Smartlog* grammar and compilation into the PostgreSQL procedural language.

2.2 Architecture of the network

Smart grids involve many heterogeneous smart devices in order to ensure a reliable, flexible and self-healing operation of the power system by updating and processing data regularly or even in real-time.

2.2.1 Architecture of a node

In this thesis, each smart device is in charge of the computation and communication at its node. Figure 2.1 presents the common architecture of nodes :

There are four main components in the architecture of a node:

Local database: It stores the node's information, such as the parameters of the information network infrastructure, the set-points values (output), the control parameters, and the measured data which are collected from local sensors or directly from the grid. The local data are part of the grid's information. This mechanism limits

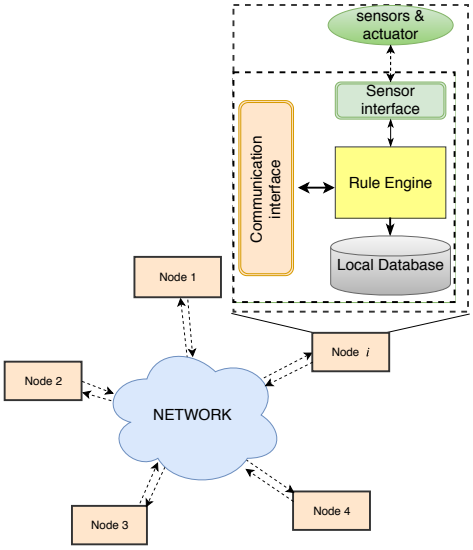


Figure 2.1: The structure of each node in the considered smart grid.

data concentration on a central server as well as accelerates the time of local access to data. Ultimately, this structure makes the smart grid become a big distributed database which is absolutely scalable.

Sensor interface: It is set up to collect and store sensor data into the local database, and transfer the output values to the actuators in order to control the active electrical components.

Rule engine: It is the most important component of the node. It supplies an environment to trigger and execute rules of local declarative programs. The main operations of the rule engine are similar to the rule-based system shown in Subsection 1.2.2.6.

Communication interface: It is in charge of the interaction with the other nodes over the communication network. Received data will be stored in the local database. In the context of this thesis, each node in the network has a unique IP address which allows a communication among nodes without errors.

2.2.2 Architecture of the network

Each node of the network possesses the same architecture. We consider an IP network in which each node can work simultaneously as both a *client* and a *server*, as shown in Figure 2.2. In this network, all nodes can provide and use resources. That allows

increasing the robustness and reducing the damage when there is a single point of failure in the system [53, 59] .

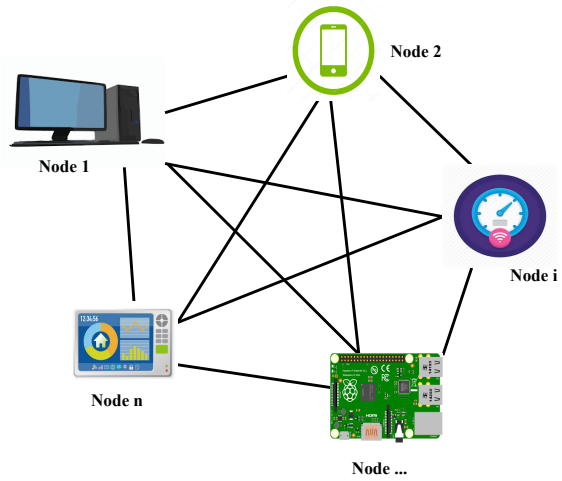


Figure 2.2: Illustration of the IP network architecture.

In the context of this thesis, we developed a system based on the available infrastructure for the information network, such as WiFi, 3G, 4G and deployed our test-cases in a microgrid. Thus, we assume that each node in the network is thoroughly capable of communicating with all others.

2.3 The *Smartlog* language

Smartlog is a rule-based data manipulation language. It is used to support declarative network data manipulation and distributed programming. *Smartlog* is designed based on Netlog, itself being an extension of Datalog. In this section, *Smartlog* is described by highlighting the differences among these three languages and its specific adaptation to the smart grid needs.

The general structure of a *Smartlog* program (P) contains three main parts shown in Listing 2.1: `data_types` definition, `initial_data` values and rules grouped in modules.

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

```
Module(data_type 2) {//rules
} ..}
```

Listing 2.1: General structure of a "Smartlog" program.

2.3.1 The `Data_types` block

Data are declared in the form of a scheme called the `data_type`. It contains the name of the `data_type`, its attributes, and the data type of each attribute. A notion of identifiers, the *key* keyword, is introduced to indicate a subset of attributes having unique values in a `data_type`. That allows optimizing data query and accelerating data access in the local database. For example, the instruction presented in Figure 2.3 is used to declare the `data_type Neighbor` in a `Data_types` block. The data_type `Neighbor` has three

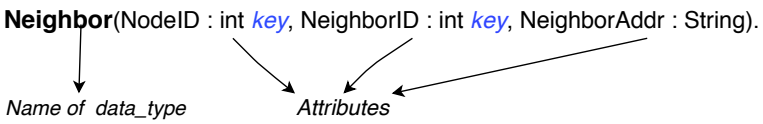


Figure 2.3: Declaration of the `data_type Neighbor`.

attributes, two of which are critical : `NodeID` and `NeighborID`. They are assigned a *key* keyword after their data type.

In smart grids, especially for AMN, the data used for distributed control can be divided into four types:

Sensor data: It is a set of data that are collected by sensors.

Parameters: It can be additional information in the communication network or constants used in the control process.

Intermediate data: It contains the auxiliary data items used in the computation process.

Output data: It contains the result of computations and normally represent control variables.

2.3.2 The `Initial_data` block

This block is not mandatory, the data stored in this block will not activate any rule in the **Module** blocks. The parameters and the output data are set up in this block, if needed. For example, the initial data for the `data_type Neighbor` declared in Figure 2.3 is written with the following `Atom`: `Neighbor(1, 2, '192.168.1.102')`.

2.3.3 Rules and Modules

Rules are the active parts of a *Smartlog* program. All rules triggered by an update or insertion of data items of a given `data_type` are regrouped in an unique `Module` which is identified by its `data_type` name. Rules will be performed sequentially in a `Module`. For example, a `Module` containing the rules triggered by the `data_type Measure`, is expressed in Listing 2.2.

```
Module(Measure) {
NeighborMeasure(j, i, v, c) :- Measure(i, v, c), Neighbor(i,j,_);
}
```

Listing 2.2: A sample "Module".

If there is an update in the `data_type Measure`, the `Module` named `Measure` will be activated and its inside rules are evaluated.

2.3.4 Syntax of the rules

The rules are in charge of defining the behaviors of each node referring to a `data_type`. The syntax of a rule in *Smartlog* is the same as in other logic programming languages: `Head : Body [terminator]`, with the *Body* part $\mathcal{B}$ of the rule $\mathcal{R}$ being the intersection of the *terms* $\mathcal{B} = \{\mathcal{B}_1, \mathcal{B}_2, ..., \mathcal{B}_n\}$. The first term of the body part relates to the identifier of its module and represents the updated/inserted data items. If all terms in the body part are verified, then the *head* part $\mathcal{H}$ will be produced.

The rule is a set of terms, each term can be an atom, a condition or an assignment and use dedicated operators, each one may involve variables. All those terms are defined below.

2.3.4.1 Variables

A set of variables in a rule $\mathcal{R}$ (including the body part $\mathcal{B}$ and the head part $\mathcal{H}$) is called $Var(\mathcal{R})$. The variables are distinguished between uninterpreted variables, denoted $(\mathbb{N}, \leq)$, and arithmetic variables, denoted $(\mathbb{R}, +, \times, \leq)$ [34]. Given a set of data I, a mapping Θ on I from $Var(\mathcal{R})$ to $\mathbb{N} \cup \mathbb{R}$ is denoted $\Theta_I : Var(\mathcal{R}) \rightarrow \mathbb{N} \cup \mathbb{R}$. We define φ_I a mapping on I from $Var(\mathcal{B})$, such that:

$$\varphi_I(\mathcal{B}) = \{\varphi_I | \varphi_I \subset \Theta_I, \forall \mathcal{B}_i, \varphi_I(Var(\mathcal{B}_i)) = \mathcal{B}_i\} \tag{2.1}$$

In a similar way, we define τ_I as a mapping on I from $Var(\mathcal{H})$, such that:

$$\tau_I(\mathcal{H}) = \{\tau_I | \tau_I \subset \Theta_I, \forall x \in Var(\mathcal{H}), \varphi_I(x) = \tau_I(x), \varphi_I = \mathcal{B}\} \tag{2.2}$$

With

$$\mathcal{B} = \bigcup_{i=1}^{n} \mathcal{B}_i \tag{2.3}$$

Variables representing attributes Variables are used to store temporary values of attributes in a scheme instance. In *Smartlog*, it is not necessary to declare the variables used in the rules. The declaration of variables in an atom is defined automatically by considering its position in the `data_type`. *Smartlog* provides many formats of data types such as *int*, *float*, *long*, *char*, *string* and *timestamp*. The symbol "_" is used to indicate that one attribute is ignored in the `data_type`.

Unlinked and linked variables In *Smartlog*, there is no random choice for a variable as in Netlog. The choice of data items is based on the relation of variables in the body part of the rule. In fact, the variables are divided in two categories: variables with assigned values (*linked variable*), and variables with no assigned values yet (*unlinked variables*). All variables in the first atom are linked variables. The unlinked variables are defined relying on at least one linked variable in the rule. If the linked variable corresponds to the whole set of *key* attributes in the `data_type`, there is only one possible value for unlinked variables. Otherwise, we must check all existing values for unlinked variables in the database. The head part needs, then, to be executed with each value

assigned to the variable. For example, let us consider the rule described in (2.4).

```
NeighborMeasure(j,i,v,c):- Measure(i,v,c), Neighbor(i, j,_).        (2.4)
```

In that example, all the variables in `Measure` are linked variables. The second Atom, `Neighbor`, has j, an unlinked variable. But j is also a *key* attribute of `Neighbor`. Therefore, the execution of the head part will be generated a quantity that depends on the number of neighbors of the node i. The `data_type` description for this example is provided in Table 2.1.

Table 2.1: `data_type` description for the rule described in (2.4).

Atom	scheme description
Measure(i, v,c)	Measure(NodeID (key), Voltage, Current).
NeighborMeasure(j, i, v, c)	NeighborMeasure(NodeID (key), NeighborID (key), Voltage , Current)
Neighbor(i, j,_)	Neighbor(NodeID (key), NeighborID (key), NeighborAddress)

Operators for variables Two main categories of operators are used for the variables in *Smartlog*:

- **Arithmetic operator:** It is in the form of $E_1 \circ E_2$, with E_1 and E_2 variables or arithmetic terms, and $\circ$ one of the standard arithmetic operators $(+, -, \times, \div)$, complex arithmetic operators (e.g. $\sin$, $\cos$, $sqrt$) or comparison operators (e.g. $>$, $<$, $\neq$, $\leq$, $\geq$, $==$) .
- **Aggreation operator:** It is in the form of $\eth(x)$, with x a variable and $\eth$ an aggregate function (e.g. min, max, avg, sum , $count$).

The rule is a set of terms which can be atom, condition term, assignment term and use dedicated operators presented below:

2.3.4.2 Atoms

An atom is in the form of $R(r_1, r_2, ..., r_n)$, where R is a name of `data_type` and $r_1, r_2, ..., r_n$ a set of variables corresponding to the data items of the attributes in the `data_type` R. For example, `Neighbor(i, j, b)` is a relational atom of the `data_type` `Neighbor`.

2.3.4.3 Condition terms

A condition term is in the form of $E_1 \diamond E_2$, where E_1 and E_2 are variables or arithmetic terms, and $\diamond$ is one of the comparison operators. For example, "$a > (b+c)$" is a condition term. A condition term will return a binary value (*true* or *false*).

2.3.4.4 Assignment terms

Assignment term is in the form of $v := E_2$ with v a variable and E_2 a variable or arithmetic terms. For example, $a := (t + 1)$ is an assignment term.

2.3.4.5 Operators for terms

The "negation" operator To negate a relational atom ($Neg_{\mathcal{R}}(I)$) we use the form $\sim R(r_1, r_2, ..., , ..., r_n)$. It is true if there is no query result in the data items at linked variables.

$$Neg_{\mathcal{R}}(I) = \begin{cases} \textbf{true} & \text{if } \forall C \in \mathbb{R}, R(\varphi(r_1), \varphi(r_2), ..., C, ..., \varphi(r_n)) \notin I \\ \textbf{false} & \text{otherwise} \end{cases} \tag{2.5}$$

The "consumption" operator It is used to delete data items of an atom having linked variables. Expressed by $\Delta^{-}_{\mathcal{R}}(I)$, we use the form $!R(r_1, ..., r_n)$ where R is a relational atom. This operator allows managing an overload of the local data storage capacities. The execution of the consumption operator with the rule R will modify I such as expressed in (2.6).

$$I := \{I \setminus \Delta^{-}_{\mathcal{R}}(I), \Delta^{-}_{\mathcal{R}}(I) = R(\varphi(r_1), \varphi(r_2), ..., \varphi(r_n)\} \tag{2.6}$$

Operators for the head part The head part contains linked variables with values assigned in the body part and defines the execution of the rule. Given J a set of incoming data via the interfaces, the result of the head part produced via the set of data $(I \cup J)$ is $\Delta^{+}_{\mathcal{R}}(I \cup J) = \tau_{I \cup J}(\mathcal{H})$. If all the terms of the body are verified, the execution of head part is launched.

The "store" operator By default, the execution of the head part is in *storing mode*, which means the results are stored into the local database. The rule is in the form: H :- $B_1, B_2, ..., B_n$. The execution of the *storing mode* of the rule ($\mathcal{R}$) is denoted by $\Phi^{\downarrow}_{\mathcal{R}}(I, J)$ and defined by:

$$\Phi^{\downarrow}_{\mathcal{R}}(I, J) = I \cup J \cup \Delta^{+}_{\mathcal{R}}(I \cup J) \setminus \Delta^{-}_{\mathcal{R}}(I \cup J) \tag{2.7}$$

The "send" operator The execution can be in *sending-mode* and expressed as: $^{\wedge}H$:- $B_1, B_2, ..., B_n$. In this case, a destination's address should be marked with the "`@`" symbol in front of the *address variable*. For instance:

$$^{\wedge}\texttt{NeighborMeasure}(\texttt{j}, \texttt{i}, \texttt{v}, \texttt{c}) : -\texttt{Measure}(\texttt{i}, \texttt{v}, \texttt{c}), \texttt{Neigbor}(\texttt{i}, \texttt{j}, @\texttt{k});$$

The execution of the *sending mode* for the rule $\mathcal{R}$ is denoted $\Phi^{\uparrow}_{\mathcal{R}}(I \cup J) = \Delta^{+}_{\mathcal{R}}(I \cup J)$.

The "store and send" operator In the *storing and sending mode*, the rule is expressed of the form: $\&H$:- $B_1, B_2, ..., B_n$. Note that the amount of local data is updated as in *storing mode* and the amount of sent data is computed as in *sending mode*.

2.3.4.6 Terminator

A rule can be terminated by a dot (".") or a semi-colon (";"). If it is a dot, the program will exit the current `Module` after the rule execution. Otherwise, the next rule in the current `Module` is triggered. Besides, the change in data is eventual. Modules may be executed in parallel, which makes the order of programs undefined. The *Smartlog* language specifically supports declarative programming for that purpose.

2.3.5 Fixpoint's semantic of a *Smartlog* program

A *Smartlog* program P has possibly many rules. With a set of incoming data J, rules in the program are triggered, and some of them are executed. The one-round execution of P in a computing node α on datasets I and J is provided by a sequence $(I^{\alpha}_i, \mathcal{P}^{\alpha}_i)_{i=0,1,...}$. With I^{α}_i and $\mathcal{P}^{\alpha}_i$ the local dataset and data to send from the node α at step i, respectively.

They are defined as follows:

$$\begin{cases} I_0^\alpha = \Phi_P^{\downarrow}(I, J) \\ I_{i+1}^\alpha = \Phi_P^{\downarrow}(I_i^\alpha, \emptyset) \text{ for } i \geq 0 \end{cases} \quad \text{and} \quad \begin{cases} \mathcal{P}_0^\alpha = \Phi_P^{\uparrow}(I \cup J) \\ \mathcal{P}_{i+1}^\alpha = \Phi_P^{\uparrow}(I_i^\alpha) \cup \mathcal{P}_i^\alpha \text{ for } i \geq 0 \end{cases} \tag{2.8}$$

The one-round computation of P in a node contains many one-round executions. The one-round computation of P in α on the set of data $I \cup J$ terminates if all its non-deterministic one-round executions converge to a fixpoint. This means that every sequence $(I_i^\alpha, \mathcal{P}_i^\alpha)$ has a limit $(I^\alpha, \mathcal{P}^\alpha\ )$ for $i \to \infty$. Such a limit is called the *one-round fixpoint* of the program P in the node α.

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

2.3.6 *Smartlog* as a support of distributed programming

Rules in a *Smartlog* program are grouped into modules which define all actions of the system with any modification of a specific `data_type`. Measured data are stored and trigger calculations in local databases. Meanwhile, intermediate data, created during the executions of the rules, support data sharing. The data transferred between nodes are in the form of `data_type`. Sent data allow triggering the next calculations in another node. For example, consider two modules in two different programs as expressed in Listing 2.3 and Listing 2.3.

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

Listing 2.3: "Module" A in node i.

```
Module(TmpC){
C(i, v, c) :- !TmpC(i, v, c), c>5;}
```

Listing 2.4: "Module" $TmpC$ in node j.

When the rule of the `Module A` in node i is executed, the `TmpC` atom is sent to node j. The `Module` of Listing 2.4 in node j is performed after receiving `TmpC`. With this mechanism, *Smartlog* can support fully distributed programming. In this context, we define the convergence of distributed *Smartlog* programming as follows [34]:

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

2.4 The grammar of *Smartlog*

The *Smartlog* grammar is defined using the meta-syntax expression, which is a formal mathematical way for specifying the syntax of languages. There are many types of meta-syntax, but in this thesis, we use the Extended Backus-Naur Form (EBNF) notation [51] to describe the *Smartlog* grammar because of its simplicity. An EBNF notation is a set of rules whose relation is defined as: `<name> = ‘‘expansion’’;`, where `<name>` is a non-terminal symbol and `‘‘expansion’’` consists of terminal or non-terminal symbols, linked together by sequencing and choices.

The terminal symbol corresponds to a single symbol or a single word that may not be decomposed. The non-terminal symbol, on the contrary, can be decomposed into other terminal or non-terminal symbols. Some additional definition of form for the EBNF are proposed in Table 2.2 and explained below:

- The expansion is expressed with square brackets [...].
- The repetition of zeros or more than one symbol is expressed with braces: {...}.
- The expansion group is indicated by parentheses (...).
- Alternative choices in an expression are separated by the “|” symbol.
- The semi-column “;” indicates the end of a rule-definition.
- The terminal symbol is enclosed in quotation marks: “...”.

An expression of the grammar of a *Smartlog* language, defined using the ENBF, is presented in Listing 2.5:

Table 2.2: EBNF main notation.

Usage	ENBF Notation
Definition	=
Concatenation	␣
Termination	;
Option	[...]
Repetition	{...}
Grouping	(...)
Terminal String	”...”
Alternative choice	|
Exception	-

```
//
//* IDENTIFIER *//
//
Digit = "1" |  "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9" | "0" ;
Upper_case = "A" | "B" | - | "Z";
Lower_case = "a"| "b" | - | "z";
Letter = Upper_case | Lower_case;
Ident = Upper_case { Letter | Digit };
Variable_ident = Lower_case { Lettre | Digit };
//
//* CONSTANT *//
//
String = "'" { Letter | Digit } "'";
Number = Digit { Digit};
Float = Number [ "." Number ] [ "E" [ "-" ] Number ];
bool = "true" | "false";
timestamp = ( "'" [ Number "-" Number "-" Number ] [ Number ":" Number [ ":" Number [
    "." Number ] ] ] "'" )| "now";
function_ident = "max"| "min" | "sum" |  "count" | "avg" | "cos" | "sin" | "tan" |
    "cot" | "acos" | " asin" | "atan" | "atan2" | "mod" | "pi" | "abs" | "power" |
    "round" | "sqrt" | "sign" | "exp" | "log" | "div" | "length" | "random" |
    "greatest" | "least" | "degrees" | "radians" ;
constant = [ "-" ] Number | [ "-" ] Float | bool | String | timestamp;
type = "int" | "float" | "boolean"  | "string" | "timestamp";
//
//* DATA_TYPEs *//
//
attribute = Ident ":" Type [ "key" ] ".";
data_type_decl = Ident "(" attribute { "," attribute } ") .";
data_type_bloc = "data types {"  { Data_type_decl } "}" ;
```

```
initial_data = Ident "(" constant { "," constant } ").";
initial_data_bloc = "initial data {" { initial_Data } "}";
//
//* EXPRESSION * //
//
unary_exp = constant | function | Variable_ident | parentized_exp | negative_exp ;
negative_exp = "-" exp;
parentized_exp = "(" exp ")";
additive_op = "+ " | " - ";
multiplicative_op  = "*" | "/" | "\" | "%";
multiplicative_exp = unary_exp { multiplicative_op unary_exp };
exp = multiplicative_exp { additive_op multiplicative_exp };
condition_op = "==" | "<>" | ">" | ">=" | "<" | "<=";
condition = exp condition_op exp;
assignment = Variable_ident ":=" exp;
address = "@";
function = function_ident "(" exp { "," exp}  ")";
//
//* RULE HEAD * //
//
head_term = constant | Variable_ident;
head =  ( [ "&"| "^" ] Ident "(" ( [ address ] head_term { "," head_term } ")" ) | (
    head_term { "," head_term } [ "," address  head_term ] {"," head_term } ")" ) ) | (
    Ident "(" head_term { "," head_term } ")" ) ;
//
//* RULE BODY *//
//
body_term = exp | "_";
atom = Ident "(" (address Variable_ident { "," body_term } ")" |body_term["," address
    Variable_ident] {" ," body_term} " )" );
literal = ( [ "!" | "~" ] atom ) | condition | assignment;
body = [ "!" | "~"] Ident "(" ( [address]  Variable_ident { "," body_term } ")"  { ","
    literal } )
                | ( body_term { "," body_term } "," address Variable_ident { ","
    body_term } ")"  { " , " literal } ) | ( ")" { ","  literal } ) ;

// * RULE *//
rule = (head)?  ":-"  body [ "." | " ;" ];
// * MODULE * //
module_bloc = "module ("  Ident ") {" rule{ rule } "}" ;
// * PROGRAM * //
```

```
Program_bloc = " program ("  Ident  ") {"  [metadata_bloc ] data_types_bloc [
    initial_data_bloc ] module_bloc{ module_bloc } "}";
```

Listing 2.5: Smartlog grammar written in ENBF.

2.5 *Smartlog*'s node prototype

The implementation of a *Smartlog* node architecture is done following the structure described in Figure 2.1. As *Smartlog* `data_types` are regular, we choose to use relational database as local database. Most relational DBMS support triggers that can be exploited for rule execution. *Smartlog* nodes are embedded devices, so it can be interesting to select a free open-source DBMS that can be instantiated on various lightweight computers such as smart meters [97] or Raspberry Pis [91] which still offers good performance. We have selected PostgreSQL [86] for local database and rule engine as it offers what we need and as there was preexisting expertise in our research group. The sensitivity of the language to the DBMS is a perspective work.

2.5.1 Rule engine

PostgreSQL provides an event-based processing called trigger, which will be automatically executed when a specified database event occurs. PostgreSQL also allows registering a procedure when a specified event takes place. The procedure can be executed *before*, *after* or *instead of* the SQL operation generating the event, and the event can be *insert*, *update*, *delete* or *truncate*. Triggers in PostgreSQL can be written in PostgreSQL Procedural Language (PL/pgSQL). They can be executed once for a query or once for every data item modified by a query (*for each row* syntax).

Smartlog rules must be executed for each actual insertion or update of data items stored in the local database. Thus, the corresponding PostgreSQL triggers are executed *after insert or update for each row*. The execution of a trigger is always part of the transaction generating the event. So, if the transaction fails at run-time, then the trigger has no effect.

A trigger will be created for each module of a *Smartlog* program and the corresponding PL/pgSQL procedure will manage the selection and execution of the rules.

2.5.2 The interfaces

A specific Java program manages the *Smartlog* nodes interfaces. This program manages data exchange among nodes using TCP/IP sockets, and acts also as the sensor interface. All incoming data (from sockets or sensors) are immediately stored in the local database, eventually triggering modules and rules. This program also listens to PostgreSQL notifications in order to initiate data communication to other nodes.

2.5.3 *Smartlog*'s compiler

A compiler is a process for translating the source code of a high-level programming language into a lower-level programming language which can be executable in the computer [71]. Errors message should be displayed if the compilation fails. The principle of the compiler is described in Figure 2.4

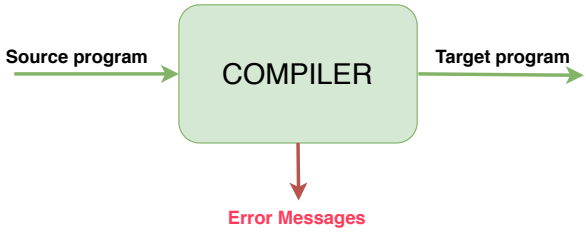


Figure 2.4: Compiler's principle.

There are many types of compilers such as Assembly, Bytecode, source-to-source, etc. In this thesis, the *Smartlog* language is compiled into PostgreSQL procedures (PL/pgSQL) as triggers installed in *Smartlog* nodes.

2.5.3.1 Compiler's workflow

In the compilation process, the grammar plays an important role. It describes the structure to analyze the input source text. The general workflow of the compilation consists of four steps, shown in Figure 2.5.

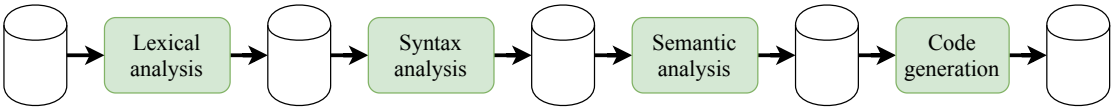


Figure 2.5: Steps of the compiler's workflow.

Let us use the expression (2.9) as an illustration support of the compiler's workflow.

$$x > a + 1 \tag{2.9}$$

Lexical analysis: It is the process of converting a sequence of input characters from source text into meaningful sequences of *lexers* or *tokens*, which are terminal symbols and are declared in the *Smartlog* grammar. For example, with (2.9), the output of lexical analysis with the grammar of Listing 2.5 is five *tokens*: `[(Letter, x), (condition_op, >), (Letter, a), (additive_op, +), (digit, 1)]`.

Syntax analysis (parsing): It is a process to analyze a sequence of tokens according to the formal grammar rule. That allows detecting any errors in the source code. The output of the syntax analysis is a parsing tree. It concretely reflects the syntax of the input language and all relationships between individual tokens. For instance, the parsing tree created after the syntax analysis of (2.9) is proposed in Figure 2.6.

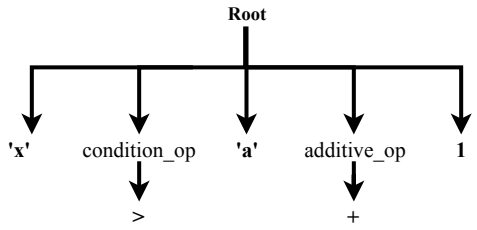


Figure 2.6: Parsing tree resulting from the syntax analysis of (2.9).

Semantic analysis: It determines the meaning of the structure of tokens and rules. The semantics of the grammar is described in detail in Subsection 2.5.3.2. This process helps detecting any semantic error in the input source text such as undeclared variables, data type incompatibilities, variable's name duplication, and so on.

Code Generation: It rewrites the semantics of a source text in the target language.

2.5.3.2 Semantic analysis of *Smartlog*

The semantics of *Smartlog* language are described with three main parts:

The `Data_types` block Each `data_type` corresponds to the creation of a table in PostgreSQL. This table consists of `data_type`'s attributes. The *key* attributes in *Smartlog* are declared as the primary keys in PostgreSQL. That is important in the data management to accelerate the access time as PostgreSQL automatically creates accelerating structures (index) on table keys. The data type conversion between *Smartlog* and PostgreSQL is presented in Table 2.3:

Table 2.3: *Smartlog* to PostgreSQL data type conversion.

***Smartlog* data type**	**PostgreSQL data type**
String	text
int	int
boolean	boolean
double	numeric
timestamp	timestamp

For example, the `data_type Neighbor` from Figure 2.3 is translated in PostgreSQL as expressed in Listing 2.6.

```
CREATE TABLE Neighbor (NodeID  int, NeighborID  int, NeighborAddr  text, PRIMARY
    KEY(NodeID, NeighborID));
```

Listing 2.6: Translation of a "data_type" in PostgreSQL.

The `Initial_data` block In *Smartlog*, it is a set of Atoms. Each atom in this block is an *insert* query. The `data_type` of these atoms must be declared before the insertion. For instance, an `initial_data`, described in Listing 2.7 will be translated in PostgreSQL as expressed in Listing 2.8.

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

Listing 2.7: Smartlog's "initial_data".

```
INSERT INTO Neighbor(NodeID, NeighborID, NeighborAddr) VALUES (1, 2, '192.168.1.102');
```

Listing 2.8: Translation of a sample "inital_data" in PostgreSQL.

The `Module` block The *Smartlog* `Module`s are translated into PostgreSQL triggers. Variables used in the `Module` have to be declared before beginning the trigger function.

Their data type is determined according to their attribute types. The trigger function returns the value held in the *new* variable. At the beginning of the function, all attributes of the first atom of the body part are assigned to *new*. The function is executed sequentially until reaching the *return new* command. For the following examples, for simplicity, the scheme is declared as `A(x, y)`, `B(x, y, z)`, and `C(x, y, z)`. A simple `Module` presented in Listing 2.9 is translated in PostgreSQL triggers procedure as expressed in Listing 2.10.

```
Module(A){
C(i, j, v) :- A(i, v), B(i, j,_).
}
```

Listing 2.9: A sample "Module".

```
CREATE OR REPLACE FUNCTION mod_A()
RETURN TRIGGER
LANGUAGE plpgsql AS
$$
DECLARE
i A.attribute1%TYPE;
v A.attribute2%TYPE;
j B.attribute2%TYPE;
BEGIN
i := NEW.x;
v := NEW.y;
-- define rule

...
RETURN NEW;
END;
$$
CREATE TRIGGER mod_A
AFTER INSERT OR UPDATE ON A
FOR EACH ROW
EXECUTE PROCEDURE mod_A();
```

Listing 2.10: Translation of the "Module" in PostgreSQL triggers.

Body part of the rule The rule will be translated from the second term of the body part to the head part. The head part defines the action of the rule which can be

translated into *insert*, *update*, or *send* data. Meanwhile, the body contains a set of terms to be evaluated. A term can be an *Atom*, a *comparison condition*, or an *assignment*.

If a literal is an atom, it's a *select* statement of unlinked variables from Atom data items to linked variables. By default, *if exist* is added before the *select* statement to ensure that the result of this statement returns a *not null* value.

If linked variables correspond to all the attributes of the primary key, the *select* statement returns a single tuple d. If not, the *select* statement returns a set of tuples. In this case, a *cursor* is set up to encapsulate the statement and fetch all the results of the query. For example, for the rule is presented in (2.10), a *select* statement of the scheme B can be translated in two cases.

$$C(i, j, v) : -A(i, v), B(i, j, \_). \tag{2.10}$$

Case 1: B has one key attribute x, `B(x, y, z)` as presented in Listing 2.11. For the variable j, which is a non-key attribute, the selection returns a unique tuple for each i. So, there is one result of the head part produced. This is expressed in Listing 2.11.

```
IF EXIST (SELECT * FROM B WHERE x=i) THEN
SELECT y INTO j FROM B WHERE x = i;
DELETE FROM C WHERE x=i AND y = j;
INSERT INTO C(x, y, z) VALUES (j, i, v);
END IF;
```

Listing 2.11: Generation query for non-key attributes.

Case 2: B has two key attributes x and y, `B(x, y, z)` as presented in Listing 2.12. In this case, there are possibly many tuples for each i. A *cursor* called Q1 is added to handle all tuples and fetch each tuple into Q1*row*. Each tuple in Q1 can produce a result in the scheme C. This is expressed in Listing 2.12.

```
IF EXIST (SELECT * FROM B WHERE x = i) THEN
OPEN Q1 FOR SELECT y FROM B WHERE x = i;
LOOP;
FETCH Q1 INTO Q1ROW;
EXIT WHEN NOT FOUND;
j = Q1ROW.y;
```

```
DELETE FROM C WHERE x = i AND y = j;
INSERT INTO C(x, y, z) VALUES (j, i, v);
END LOOP;
CLOSE Q1;
END IF;
```

Listing 2.12: Generation quey for key attribute.

The operators of an Atom can be a "negative" ($\sim$) and a "consumption" operator (!). The "negative" operator ($\sim$) is translated in PostgreSQL query *if not exist*. For example, the translation of the rule (2.11) is shown in Listing 2.13.

$$C(i, i, v) : -A(i, v), \sim B(i, \_, \_). \tag{2.11}$$

```
IF NOT EXIST (SELECT * FROM B WHERE x = i) THEN
DELETE FROM C WHERE x=i AND y = i;
INSERT INTO C(x, y, z) VALUES (i, i, v);
END IF;
```

Listing 2.13: Generation query with the negation operator.

The "consumption" operator (!) is used to delete the atom for linked variables. It is translated into the *delete* statement. For example, the rule (2.12) is translated in PostgreSQL as shown in Listing 2.14.

$$C(i, j, v) : -A(i, v), !B(i, j, \_). \tag{2.12}$$

```
IF EXIST (SELECT * FROM B WHERE x = i) THEN
SELECT y INTO j FROM B WHERE x = i;
DELETE FROM C WHERE x = i  AND y = j;
INSERT INTO C(x, y, z) VALUES (i, j, v);
DELETE FROM B WHERE x=i AND y=j;
END IF;
```

Listing 2.14: Generation query with the consumption operator.

If a term is a condition, the expression in *Smartlog* is translated in a *if-then* statement in PostgreSQL. For example, the condition term of rule (2.13) is translated in Listing 2.15

in PostgreSQL.

$$C(i, i, v) : -A(i, v), v > 380. \tag{2.13}$$

```
IF(v > 380) THEN
DELETE FROM WHERE x = i  AND y = i;
INSERT INTO C(x, y, z) VALUES (i, i, v);
END IF;
```

Listing 2.15: Generation query with a condition.

If a term is an assignment operator (:=), it is translated to (=). *Smartlog* supports many functions that are identical in PostgreSQL, such as sin , cos , log , *sqrt*, except for the aggregate function. It must be noticed that the aggregate function divides the rule into two parts (in before and after the aggregate function) such that the terms after the aggregate function can only intervene in linked variables which are determined at the starting of the aggregate computation, such as i, v in rule (2.14).

$$C(i, v, k) : -A(i, v), B(i, j, t), k := sum(t). \tag{2.14}$$

The result of the aggregate function is computed incrementally for all tuples of a query. An initialization for the results of *sum* and *count* is required before executing the query. For example, a *sum* function in rule (2.14) is translated in PosgreSQL as expressed in Listing 2.16.

```
k := 0;
IF EXIST (SELECT * FROM B WHERE x = i) THEN
OPEN Q1 FOR SELECT z FROM B WHERE x = i;
LOOP;
FETCH Q1 INTO Q1ROW;
EXIT WHEN NOT FOUND;
t = Q1ROW.z;
k = k+t;
END LOOP;
CLOSE Q1;
DELETE FROM C WHERE x = i AND y = v;
INSERT INTO C(x, y, z) VALUES (i, v, k);
END IF;
```

Listing 2.16: Generation query with an aggregate function.

Head part of the rule There are three possible actions in the head part: *store*, *send*, and both *store and send*. By default, if there is no option in the head part, the result will be stored into the local database. It is compiled into a *delete* and *insert* statement. This is shown in the previous examples.

In the case of the "sending" operator "^", data are sent to another node. The compilation program will raise a notification with *pg_notify()* to the interface, in order to activate the communication. The notification is packed using the JSONObject format, including the target address, and the result of the head part. As an illustration, the compilation of rule (2.15) is proposed in Listing 2.17.

$$^{\wedge}C(i,j,v) :- A(i,v), B(i,j,@k). \tag{2.15}$$

```
IF EXIST (SELECT * FROM B WHERE x = i) THEN
SELECT z into k FROM B WHERE x = i) ; -- choose target address
data = json_build_object( 'x' , i, 'y', j, 'z', v);
notification = json_build_object(
'address', k,
'scheme', 'C',
'data', data);
perform pg_notify('q_event', notification ::text) ;
END IF;
```

Listing 2.17: Generation query with a sending operator.

In case the option is "&", the rule is translated by combining the compilation of two actions: storing data and sending data.

Terminator If a rule terminated by a dot ("."), the *return new* command is added just after the compilation of the current rule to break out of the trigger function. Otherwise, a rule is terminated with a semi-colon (";"). In that case, the trigger function continues to execute other queries.

2.5.3.3 Environment and tools for building compilers

In this work, in order to compile the *Smartlog* to a PostgreSQL procedural language, we mainly use three tools: ANTLRv4, JAVA, and POSTGRESQL. ANTLRv4 provides

an environment to define the *Smartlog* grammar, transforms the grammar into *Parser* and *Lexer* classes in Java, and supports APIs for analyzing the lexicon and syntax of the input source text.

On that basis, the semantics of the *Smartlog* language are developed and compiled in PostgreSQL with JAVA. The output program is validated in the PostgreSQL environment. We suggest referring to [52] for details on the compiler's implementation.

Figure 2.7: Environments and tools used for the compiler.

2.6 Conclusion

In this chapter, we present a rule-based data manipulation language called *Smartlog* and the architecture network of the *Smartlog* nodes. *Smartlog* is developed to be a high-level reactive and declarative language for distributed data management. The improvement of the syntax of this language and its functionalities are compared to the preexisting declarative languages (Netlog and Datalog). Moreover, the *Smartlog* grammar and distributed fixpoint semantic are also presented. A prototype exploiting PostgreSQL as the local database has been realized in which we proposed to compile *Smartlog* programs into PostgreSQL triggers. The translation and syntax are illustrated through various examples.

Chapter 3

Methodology for easy distributed rule-based programming

3.1 Introduction

There exist various applications for control and management of power systems, the associated algorithms are traditionally deployed in a centralized way. However, along with the development of smart grids, a centralized implementation can no longer be sufficient. It shows many shortcomings, such as high computation and communication costs, a single point of failure, etc. [38].

Besides, with the current development of grids infrastructures, the computing units are almost located everywhere in the network. These enormous available resources can participate in the management of the power system to cope with the issue of scalability. A distributed implementation of the algorithms over the entire system seems to be more efficient because it deals with the imminent problems of algorithms while ensuring to replace the conventional centralized controller [87]. However, there are still some drawbacks of the current distributed implementations that restrain its deployment in reality (that is shown in Subsection 1.2.2.3). The slow convergence speed and the neglected communication delay lead to new issues in distributed control and management, which possibly will violate some boundaries of the system stability and impact the voltage and frequency quality. Besides, there exist a large variety of centralized algorithms for the control and management of power systems compared to distributed algorithms. One

of the major challenges of distributed algorithms implementation is the lack of high-level programming abstraction. In addition, the control and management of smart grids in real-time requires dealing immediately with changes in the power system and easy deployment in a large telecom network. Thus, an approach is needed to combine the advantages of both the centralized and distributed paradigms while ensuring the response to any change in the system as fast as possible.

For those reasons, in this thesis, we aim at developing a methodology for centralized programming and distributed execution (CPDE). Firstly, we aim at abstracting smart grids systems in the form of a distributed database system, in which a rule-based distributed data manipulation language, called *Smartlog*, is in charge of the data management. *Smartlog* is suitable for real-time reaction, the scalability of the system, and distributed programming support. In Chapter 2, we described the architecture of the distributed database network dedicated to smart grids as well as the development of the *Smartlog* language. In this chapter, we focus on the CPDE methodology, which helps sharing the computational load over the available computing resources with a predefined data allocation. It allows minimizing the possibility of overload in the central servers. Secondly, the computation over the network are performed in a reactive manner, based on changes of the system. This restricts redundant communications and computations. Finally, the methodology provides an easy and simple principle for algorithms implementation in a distributed system.

The CPDE method approaches the *distributed Smartlog* programming with the principle presented in Figure 3.1. A *centralized Smartlog* program (which is executed in a single machine) is exploited into multiples *distributed Smartlog* programs, which are implemented in computing nodes (multiple machines) in the telecom network. These distributed programs cooperate to accomplish centralized computations. Nevertheless, it is still a semi-automatic distributed programming method because it is deployed based on the data distribution, which is currently described manually. The automation of the data distribution is a research perspective.

In this chapter, the methodology is presented following this plan: Section 3.2 provides a model to describe the data distribution in the smart grid. The detail of the CPDE methodology is presented in Section 3.3. In Section 3.4, the behavior of the methodology is illustrated and validated on elementary test cases. A conclusion is finally drawn.

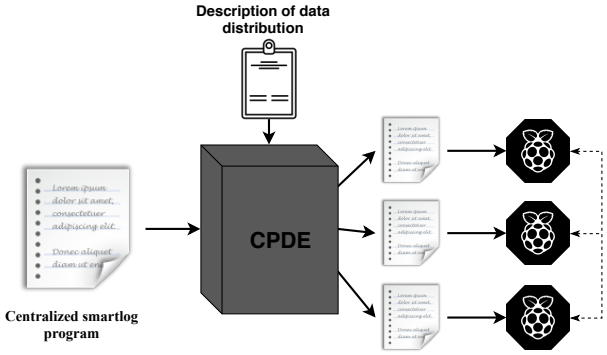


Figure 3.1: Principle of the CPDE methodology.

3.2 Data distribution

The data, which are collected by sensors and sent to actuators, are naturally scattered in the smart grid according to horizontal fragmentation. Meanwhile, the intermediate data, serving for intermediate transformations, can be located anywhere in the telecommunication network, as shown in Figure 3.2. Data in this context are not concentrated in one machine as in the centralized paradigm. That is why programming in a distributed manner is more difficult than in a centralized one. In order to perform computations, data may need to be gathered from many locations. In this section, we present the data fragmentation and allocation model to facilitate data access.

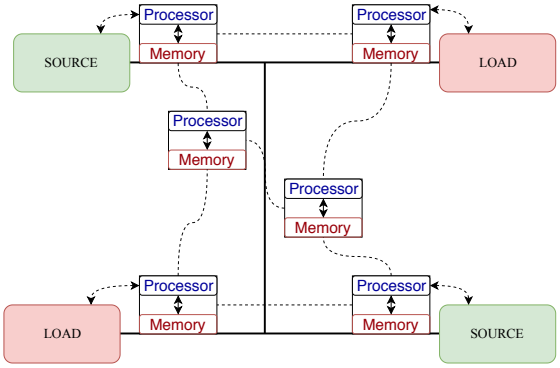


Figure 3.2: Principle of the data distribution in smart grids.

3.2.1 Fragmentation and allocation of the data types

The design of the distributed database allows each relation scheme R to be divided horizontally, vertically, or in a hybrid manner. Each part of the data is called a fragment. As presented in Subsection 1.2.2.4, for horizontal fragmentation, each fragment holds the whole attributes of the original `data_type`, but it only contains a part of the global

data record. The conjuncture of global data scheme is expressed as:

$$DT_G = \bigcup_i DT_i \tag{3.1}$$

For a vertical fragmentation, each fragment holds several attributes including the key one of the data scheme, and all data records of its attributes. The conjuncture of the global data scheme based on common key attributes is expressed as:

$$DT_G = \bowtie_i DT_i \tag{3.2}$$

For example, a vertical fragmentation of `data_type A(x, y, z)` can be `A1(x, y)` and `A2(x, z)`.

In the description of the data distribution, each scheme of a fragment is in charge of describing the properties of its dataset. For a vertical fragmentation, each fragment describes a subset of attributes of a `data_type`. In the case of horizontal fragmentation, we use conditions on identifiers (key attribute of `data_type`) to distinguish fragments. The description of the data allocation allows determining which and where data are stored.

Sensing data and control data are prioritized to be stored nears sensors and actuators in the smart grid. Meanwhile, intermediate data can be allocated anywhere and based on the optimal data placement process. The design of the optimal data placement is not the focus of this thesis. Herein, we assume that it has been well specified upstream. A model describing the data fragment is proposed instead. Besides, in the principle of the distributed database design [69], data can be located in one or more locations, but for this work, data duplication has not been considered.

3.2.2 Syntax of the data distribution

The information of all participants in the network should be declared for communication through an IP mapping table, called `IPmap`. This helps mapping the name of a data location into its network address. The information of the `IPmap` is assigned to the `initial_data` block in the *distributed Smartlog* programs.

The description of the data distribution (fragmentation) is declared as an extension of the `data_types` block (D). Each `data_type`'s description contains its original `data_type`

$(D_i|D_i \subset D\ )$ with its attributes $(A_i|A_i \subset D_i)$ and its data fragments $(F_i|F_i \subset D_i)$ with $D_i = \{A_i, F_i\}$. This description is considered as an extension of the *Smartlog* grammar and is written using the EBNF notation, as in Listing 3.1.

```
//*IPMap*//
ipmap = String ":" String ";" ;
ipmap_bloc = "IPmap" "{" ipmap { ipmap} "}" ;
//*DATA_FRAGMENT_BLOC*//
  condition_bloc = "Conditions" ";" condition {"," condition} ";" ;
  attributes_bloc = "Attributes" ":" Ident {"," Ident} ";" ;
  location_bloc = "Location" ":" String {"," String} ";" ;
  fragmentation = "fragment" ["(" Ident ")"] "{"
                  [condition_bloc] [attributes_bloc] location_bloc "}" ;
  data_fragment_bloc = data_type_decl "{" fragmentation { fragmentation} "}." ;
  //*DATA_DISTRIBUTION_BLOC*//
  data_distribution_bloc = ipmap_bloc data_fragment_bloc {data_fragment_bloc}
```

Listing 3.1: Description of the data distribution in ENBF.

For each fragment $(F_{ij}|F_{ij} \subset F_i)$, we define three parts: the conditions C_{ij}, the attributes A_{ij} and the location L_{ij}, $F_{ij} = \{L_{ij}, A_{ij}, C_{ij}\}$. The location part has to be specified, while it is not mandatory for the conditions and the attributes part. The conditions part defines which identifiers of the `data_type` belong to the considered fragment. In the case of a horizontal data fragmentation, the condition part C_{ij} should be declared. Otherwise, in the case of a vertical fragmentation, the attributes part A_{ij} describes the attributes of the original `data_type` stored in the fragment. For a hybrid fragmentation, the three parts must be declared. By default, if there is no declaration of the attributes part A_{ij}, the fragment holds all attributes of the original `data_type` $A_{ij} = A_i$. For example, the `data_type` A, declared in a *centralized Smartlog* program, is proposed in Listing 3.2.

```
Data_types{
A(ValueA1 : int key, ValueA2 : float, ValueA3 : float}.}
```

Listing 3.2: Description of a sample "data_type" A.

The description of the `data_type` A with two fragments located in the sites $S1$ and $S2$ is proposed in Listing 3.3.

```
Data_types{
```

```
A(Value1 : int key, Value2 : int , Value3 : int) {
  Fragment(A1) {
    Conditions : Value1>10;
    -- Attributes :
    Location : 'S1';}
  Fragment(A2){
    Conditions : Value1<=10;
    --Attributes:
    Location : 'S2';}
  }.
}
```

Listing 3.3: Description of a sample data_type A with two locations, $S1$ and $S2$.

The description of the data distribution of the `data_type` in Listing 3.3 is simplified in Listing 3.4 with `x, y, z` corresponding to the first, second and third attribute of the `data_type`.

```
A(x, y, z) = A1(x>10)@'S1' ⋃ A2(x<=10)@'S2'.
```

Listing 3.4: Simple form for the description of the "data_type" A.

3.3 Program distribution

As aforementioned, the CPDE method automatically deploys a *centralized Smartlog* program (which is executed in a single machine) into multiples *distributed Smartlog* programs. These distributed programs cooperate to perform the same task as the centralized one. Based on the description of the data distribution, the CPDE methodology analyzes a set of rules in a *centralized Smartlog* program (called the input of the process) into sub-rules corresponding to each location (the output of the process). Some critical points in the methodology should be satisfied:

- The *centralized Smartlog* program is supposed to run without error;
- The communication network is assumed to be reliable;
- The centralized computation is executed precisely in a distributed manner and ensures that the behaviors are the same in both implementations;

- The number of communications and transferred data must be minimized, limiting redundant communications over the network.

In this section, we explain how to distribute the calculations over the distributed network (i.e. how to deal with rules distribution) and how to auto-generate a set of collaborating *distributed Smartlog* programs equivalent to the centralized one. The basic algorithm for the CPDE methodology is proposed in Figure 3.3, which contains three main steps: rule rewriting, rule distribution and program generation.

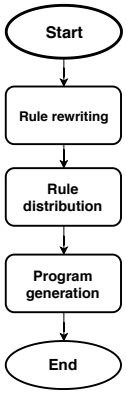


Figure 3.3: Synoptic of the CPDE methodology.

3.3.1 Rule rewriting

The objective here is to rewrite the atoms of a rule according to their fragmentation. The main ideas of this step are:

1. To develop the original rule based on the description of the data distribution. The centralized rule is expressed over the global `data_types`, and the rewritten rule should be expressed over the local fragments of the `data_types`.

2. To deal with the problem of aggregate function, if it exists inside the rule. As mentioned in Chapter 2, data aggregation is considered as a particular case in the *centralized Smartlog* program, in which the rule is divided in two parts (before/after the aggregate function) so that unlinked variables before the aggregate function cannot be accessed by terms after it.

Some symbols should be presented to simplify the description of the method. We symbolize the set of rules in a program as $\mathcal{R}$, $\mathcal{R}_i$ being the i^{th} rule ($\mathcal{R}_i \subset \mathcal{R}$). The set

of terms in the body and the head part of the i^{th} rule are respectively $\mathcal{B}_i$ and $\mathcal{H}_i$, $\mathcal{R}_i = \{\mathcal{H}_i, \mathcal{B}_i\}$. The j^{th} term in a body part is called $\mathcal{B}_{ij}$. A term can be a relational atom a_k, a condition c_j or an assignment s_j, $\mathcal{B}_{ij} \in \{a_j, c_j, s_j\}$.

3.3.1.1 Principe and algorithm

The general principle for the rule rewriting is to use fragments to replace the global `data_types` by their reconstruction. We proceed to evaluate all atoms of a rule from the body part to the head part.

For a horizontal fragmentation, where fragments are defined by applying a condition to one or more attributes participating in keys, the reconstructions of an atom are done by an union of fragments. Let us consider rule (3.3) with three atoms corresponding to three descriptions of the data distributions `A, B, H`, which are declared in Listing 3.5. The underlined attribute indicates the key attribute of the `data_type`s.

$$R_1 : H(m,k,t) : -A(m,n,t), B(m,\_,k). \tag{3.3}$$

```
H(x, y, z) = H1@'S3';
A(x, y, z) = A1@'S1';
B(x, y, z) = B1(x>10)@'S1' ⋃  B2(x<=10)@'S2';
```

Listing 3.5: Description of the "data_type" distribution of A, B and H.

In this example, `H` and `A` are not fragmented but `B` is fragmented horizontally. The rule (3.3) is rewritten in two rules corresponding to two fragments of the global `data_type` `B` as follows:

$$\begin{cases} H1(m,k,t) : -A1(m,n,t), B1(m,\_,k), m > 10; \\ H1(m,k,t) : -A1(m,n,t), B2(m,\_,k), m <= 10; \end{cases} \tag{3.4}$$

For a vertical fragmentation, where fragments are defined by listing the included attributes (only the key attribute is a common point to all fragments), the reconstruction of the global `data_type` is done by combining fragments. Using rule (3.3) again and the data distribution description of Listing 3.5, `B` is fragmented vertically as in Listing 3.6.

```
B(x, y, z) = B1(x, y)@'S1' ⋈  B2(x, z)@'S2';
```

Listing 3.6: Vertical fragmentation description of B.

Thus, rule (3.3) is rewritten as:

$$H1(m,k,t) :- A1(m,n,t), B1(m,\_), B2(m,k); \tag{3.5}$$

There is one rewritten rule for rule R_1 in which the global `data_type B` is replaced by a set of joint `B1` and `B2`. If the `data_type` of the atom is a hybrid fragmentation, the same as the vertical fragmentation is done.

For the head part of the rule $\mathcal{H}_i$, after rewriting the body part, we duplicate the rule for each fragment of the head part. If the head part is a non-vertical fragmentation, the condition of each fragment is added at the end of the body before replacing the head part by the atom of each $\mathcal{H}_i$'s fragment. For example, the head part is fragmented vertically as in Listing 3.7.

```
H(x, y, z) = H1(x>5)@'S2' ⋃  H2(x<=5)@'S3';
```

Listing 3.7: Vertical fragmentation of the head part.

Rule (3.5) is rewritten according to the data fragmentation of the head part as (3.6).

$$\begin{cases} H1(m,k,t) :- A1(m,n,t), B1(m,\_), B2(m,k), m > 5; \\ H2(m,k,t) :- A1(m,n,t), B1(m,\_), B2(m,k), m <= 5; \end{cases} \tag{3.6}$$

The terminator of a rewritten rule is the same as that of the original rule. For each intermediate rule that supports data transfer, the terminator is defined by a semicolon (";"). The detail of the rule rewriting process according to the data fragmentation is presented in Algorithm 1.

3.3.1.2 Aggregate function

An aggregate function f helps to summarize information over the network [18], which takes a set of elements I and produces an output O. It is defined as:

$$f : N^I \longrightarrow O \tag{3.7}$$

The *Smartlog* language also supports aggregate functions such as *min, max, sum, count*, and *avg* like other rule-based languages (e.g. Netlog). Most of the aggregate functions can be computed incrementally, even in distributed systems. An aggregate function f

Algorithm 1 Rule rewriting.

```
1: procedure RULEREWRITING(rule R_i)
2:     for B_ij in B_i do
3:         if B_ij instantOf Atom then
4:             get D_j from a_j
5:             D_jk ← D_j, a_jk ← {D_jk, a_j}
6:             if D_j is not a horizontal fragmentation then
7:                 Group of juncture D_jk of D_j
8:                  Replace a_j by set of junctures a_jk (a_jk ⊂ SA)
9:             else Replace a_i by set of a_jk(a_jk ∈ SA)
10:            end if
11:            if j == 0 then
12:                for a_jk in SA do
13:                    Create a new Rule R_ij, add a_jk to R_ij
14:                    add R_ij to list of Rule (LR|R_ij ⊂ LR)
15:                end for
16:            else
17:                for a_jk in SA do
18:                    for R_ij in LR do
19:                        if C_{a_jk} ∩ C_{R_ij} ≠ ∅ then
20:                            add a_jk into R_ij
21:                        end if
22:                    end for
23:                end for
24:            end if
25:        else
26:            for R_ij in LR do
27:                if C_{B_ij} ∩ C_{R_ij} ≠ ∅ then
28:                     add B_ij into R_ij
29:                end if
30:            end for
31:        end if
32:    end for
33:    for F_ij in H_i do
34:        D_ij ← F_ij, a_ij ← {D_ij, H_i}
35:        if ∃c_ij ∈ F_ij then
36:             add c_ij into R_ij
37:        end if
38:        Add a_ij to R_ij
39:    end for
40: end procedure
```

that can be decomposed, is defined by:

$$f = f_1 \diamond f_2 \tag{3.8}$$

With f_1 an identity function and f_2 a self-decomposable aggregate function. The aggregate functions in *Smartlog* are rewritten following equations (3.9) to (3.13) [41].

$$\begin{cases} SUM(a) = a \\ SUM(A \oplus B) = SUM(A) + SUM(B) \end{cases} \tag{3.9}$$

$$\begin{cases} MIN(a) = a \\ MIN(A \oplus B) = MIN(MIN(A), MIN(B)) \end{cases} \tag{3.10}$$

$$\begin{cases} MAX(a) = a \\ MAX(A \oplus B) = MAX(MAX(A), MAX(B)) \end{cases} \tag{3.11}$$

$$\begin{cases} COUNT(a) = 1 \\ COUNT(A \oplus B) = COUNT(A) + COUNT(B) \end{cases} \tag{3.12}$$

$$\begin{cases} AVERAGE(A) = SUM(A)/COUNT(A) \end{cases} \tag{3.13}$$

A disjoint union of a subset is denoted $\oplus$. In the case of a vertical data fragmentation, elements of the set $(A \oplus B)$ are located in the same location. The output of the aggregate function is computed easily with local queries. As an illustration, the rule (3.14) presents an example of an aggregate function.

$$R_2 : H(m, k, t) : -A(m, n, t), B(m, l, \_), k := max(l); \tag{3.14}$$

The rule (3.14) is rewritten as rule (3.15), and the aggregate function is not decomposed in this rule because all l values are in the same location.

$$R_{21} : H1(m, k, t) : -A1(m, n, t), B1(m, l), B2(m, \_), k := max(l); \tag{3.15}$$

In the case of a horizontal data fragmentation, the set of elements is in different locations. The output of an aggregate function is computed based on the results of all self-decomposable functions. In detail, each rewritten rule is split into two parts.

1. The first part holds the terms from the beginning of the body part to the self-decomposable function.

2. The second part expresses the identity function and holds the rest of the rule.

Using again rule (3.14) and the horizontal fragmentation of the `data_type B`, as provided in Listing 3.5, the rewritten rule (3.16), according to the data fragmentation, has to be modified into rule (3.17) according to the aggregate function. It is noted that *max* function in rule R_{31} and R_{32} is an aggregate function, but in rule R_{33_1} and R_{33_2}, it is a comparison operator.

$$\begin{cases} R_{21} : H1(m,k,t) :- A1(m,n,t), B1(m,l,\_), k := max(l); \\ R_{22} : H1(m,k,t) :- A1(m,n,t), B2(m,l,\_), k := max(l); \end{cases} \tag{3.16}$$

$$\begin{cases} R_{31} : Tmp1(m,k1,t) :- A1(m,n,t), B1(m,l,\_), k1 := max(l); \\ R_{32} : Tmp2(m,k2,t) :- A1(m,n,t), B2(m,l,\_), k2 := max(l); \\ \\ R_{33_1} : H1(m,k,t) :- !Tmp1(m,k1,t), !Tmp2(m,k2,t), k := max(k1,k2); \\ R_{33_2} : H1(m,k,t) :- !Tmp2(m,k2,t), !Tmp1(m,k1,t), k := max(k1,k2); \end{cases} \tag{3.17}$$

The result of the first part is generated by queries in each local database and saved in a temporary `data_type` (e.g. `Tmp1` and `Tmp2`). These temporary `data_types` are gathered to produce the output of the aggregate function. Data items of each temporary `data_type` possibly trigger the calculation of the output. That is why there are many rules generated for the second part. These data of temporary `data_type` are deleted once the output is produced. The deletion of the temporary data-items helps release the working space during the rule execution as well as ensure the accuracy of the result in the distributed computing paradigm. So, the consumption operator ("!") is added in front of these atoms in rule R_{33_1} and R_{33_2}.

3.3.2 Rule distribution

After determining which rules can be incurred from the original rule $\mathcal{R}_i$, the next step is to analyze these rules according to locations and decide the data transferred among locations. In this step, each rule $\mathcal{R}_{ij}$ in the set of rewritten rules LR ($\mathcal{R}_{ij} \subset LR$) can possibly be separated into multiple rules according to fragments locations. In this section, we present the steps of the optimal rule distribution.

3.3.2.1 A rule in the form of a cyclic graph

The main purpose of this step is to specify the links existing among the terms in a rule. We use a directed (cyclic) graph $\mathcal{G}(\mathcal{V}, \mathcal{E})$, where $(\mathcal{V}, \mathcal{E})$ is a set of vertices and edges of the graph respectively, indicating the link between the terms in the rewritten rule $\mathcal{R}_{ij}$.

The construction of the graph $\mathcal{G}$ represents the possible execution orders of a rule, in which the rule can be performed exactly. The orders of the rule execution are determined by passing the values from linked variables (whose value are known) to unlinked ones. The first term of the body part of a rule is the only one that starts with all its variables linked. Thus, all paths in the graph start with this term and end with the head part of the rule. We analyze all variables of the terms in the rule in the reverse order (from $\mathcal{H}_i$ to $\mathfrak{a}_{i0}$) to constitute edges. Each term (except the condition one) which can create new variables can be a vertex of the graph $\mathcal{G}(\mathcal{V}, \mathcal{E})$. As the body of a *Smartlog* rule is a conjunction of terms, all these terms are commutative. The commutativity of the terms possibly leads to producing cycles in the graph.

For example, the directed cyclic graph constituted for rule (3.5) is shown in Figure 3.4.

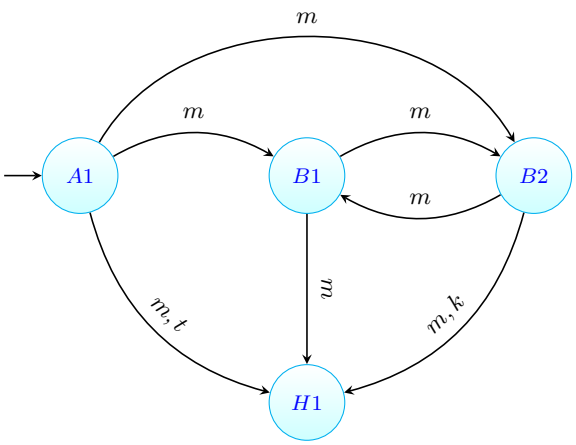


Figure 3.4: Intial cyclic graph for rule (3.5).

3.3.2.2 Covering irredundant directed acyclic graphs

The initial graph represents data paths among terms, which possibly contains cycles. In practice, cyclic data paths cannot be executed; thus, covering directed acyclic graphs have to be identified. For example, by deleting one by one the edges of a cycle in the

cyclic graph of Figure 3.4, we generate two possible acyclic graphs shown in Figure 3.5 and Figure 3.6.

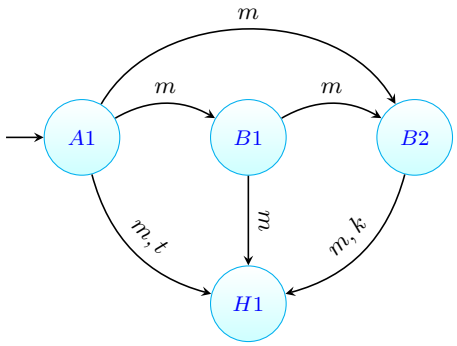


Figure 3.5: First acyclic graph for rule (3.5).

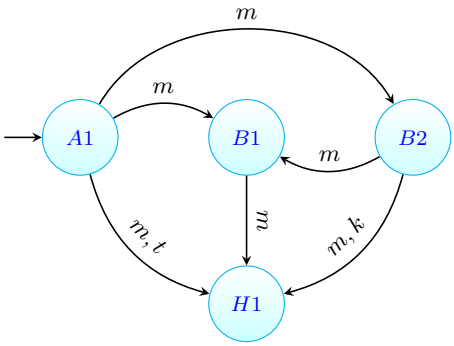


Figure 3.6: Second acyclic graph for rule (3.5).

In these graphs, there are several data paths to transfer the same value from one term to another one potentially causing useless communications. Thus, it is essential to eliminate redundant edges. The irredundant graph $\mathcal{M}(\mathcal{V}, \mathcal{E}')$ is a subset of the directed graph $\mathcal{G}$ which has all vertices covered with such a number of edges that $\mathcal{E}' \subset \mathcal{E}$ and the execution order of the rule is not changed. Note that this problem is not about finding a minimum spanning tree. Thus, methodologies such as optimum branchings [25] and Fibonacci heap [28] cannot be applied. Before defining a redundant edge in a directed graph, we define an indirect edge:

Definition 3.1. Let us consider a directed graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, with $\mathcal{V}$ a set of vertices, and $\mathcal{E}$ a set of edges. Let us consider a direct edge $\mathfrak{e}_i =< u, v >$, with $\mathfrak{e}_i \in \mathcal{E}$, head$(\mathfrak{e}_i) = u$, and tail$(\mathfrak{e}_i) = v$. A *path* from the vertex q to the vertex t, called $\mathfrak{p}_{q \to t}$, is a set of edges so that:

$$p_{q \to t} = \{ \{e_1, ..., e_k\} | \forall e_i \in \mathcal{E}, \text{head}(\mathfrak{e}_1) = q, \text{tail}(\mathfrak{e}_k) = t, \forall i \in [1..k-1], \text{tail}(\mathfrak{e}_i) = \text{head}(\mathfrak{e}_{i+1}) \} \quad (3.18)$$

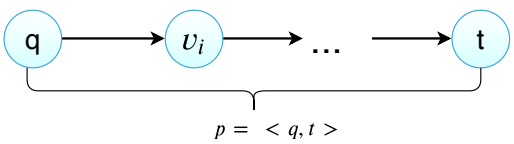


Figure 3.7: Illustration of a path p from the vertex q to the vertex t.

Definition 3.2. An indirect edge from the vertex q to the vertex t is $\mathfrak{ie}$, if it exists a path from q to t so that the number of edges in the path (k_p) is greater than 1.

$$\mathfrak{ie}_{q \to t} = \{p_{q \to t} | \exists p \subset \mathcal{E}, k_p > 1\} \tag{3.19}$$

Definition 3.3. An edge from the vertex q to the vertex t belonging to the set $\mathcal{E}$ is called a redundant edge $\mathfrak{re}$ if it exists an indirect edge from the vertex q to the vertex t.

$$\mathfrak{re}_{q \to t} = \{p_{q \to t} | p_{q \to t} \in \mathcal{E}, \exists \mathfrak{ie}_{q \to t}\} \tag{3.20}$$

For example, the irredundant graph of Figure 3.5 and Figure 3.6 are shown in Figure 3.8 and Figure 3.9:

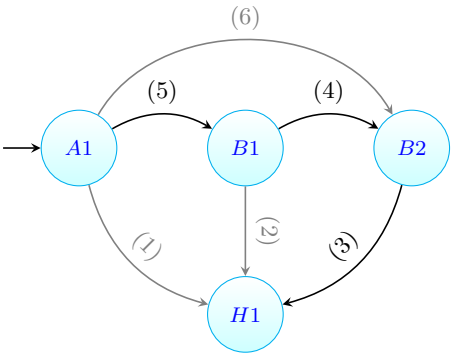


Figure 3.8: Acyclic irredundant graph of Figure 3.5.

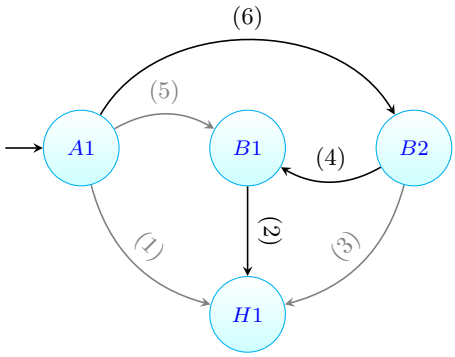


Figure 3.9: Acyclic irredundant graph of Figure 3.6.

In Figure 3.8, the first edge (1) from the vertex $A1$ to the vertex $H1$ is a redundant edge because there is an indirect edge from $A1$ to $H1$ through the edges (5), (4) and (3). For the same reason, edge (2) and edge (6) are also redundant edges because it exists an indirect edge from $B1$ to $H1$ and $A1$ to $B2$. In the same way, the edges (1), (3), and (5) are redundant in Figure 3.9.

3.3.2.3 Decisions to transfer data-items

The elimination of redundant edges requires rerouting of data items transitioning in the acyclic graph so that the necessary data are still transmitted properly to the destination (the vertex contains the head part). To decide how data items are transferred from one vertex (i.e. A) to another vertex (i.e. B), all variables of the predecessor vertices are taken into account. If a variable in A appears on a set of variables in A's successor then this variable is transferred. In case vertex A has more than one edge starting from

it, all the key variables are transferred even if it does not appear in A' successor. For example, the transferred data-items for the acyclic irredundant graphs of Figure 3.5 and Figure 3.6 are illustrated in Figure 3.10 and Figure 3.11.

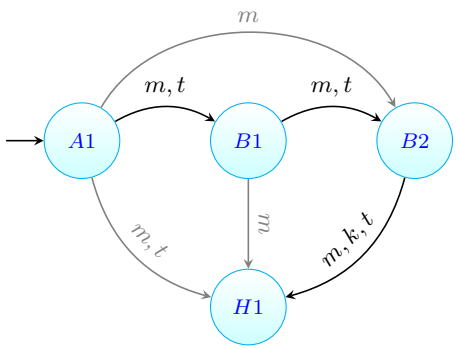


Figure 3.10: Rerouting data items in the first acyclic irredundant graph from Figure 3.5.

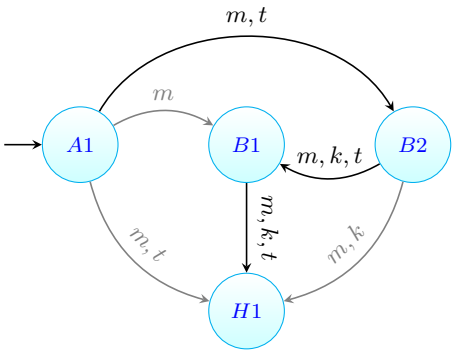


Figure 3.11: Rerouting data items in the second acyclic irredundant graph from Figure 3.6.

3.3.2.4 Location of terms

The location of each vertex must be determined before deciding the communication. The location of the atom-containing vertex can be easily known in the description of the data distribution. For instance, in rule (3.5), $A1$ and $B1$ are co-located in the site $S1$, $B2$ in the site $S2$ and $H1$ in the site $S3$, as shown in Figure 3.12 and Figure 3.13 where the red dashed rectangles indicate the locations.

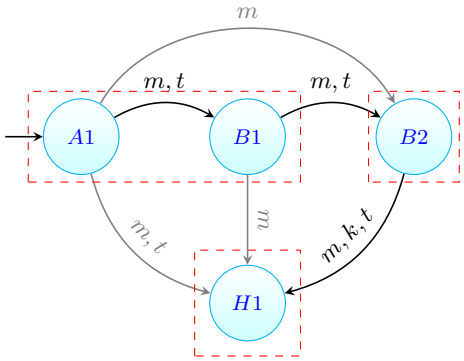


Figure 3.12: Data location in the first acyclic irredundant graph from Figure 3.5.

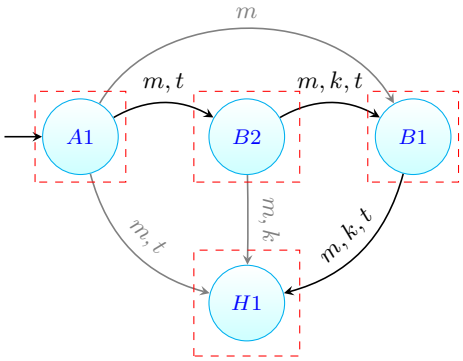


Figure 3.13: Data location in the second acyclic irredundant graph from Figure 3.6.

Meanwhile, the location of non-atom vertices (an *assignment term*) is prioritized to co-located as soon as possible within an atom-containing vertex location. The possible positions of a non-atom vertex (NA, unknown location) with regard to an atom vertex position (At, known location) are illustrated in Figure 3.14 to Figure 3.17.

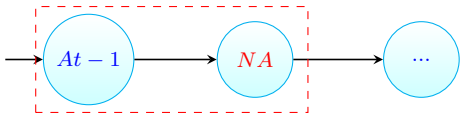


Figure 3.14: One atom vertex.

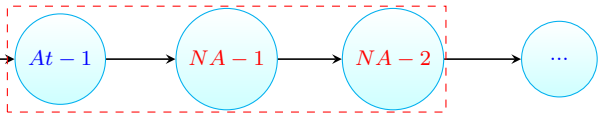


Figure 3.15: Sequence of non-atom vertices.

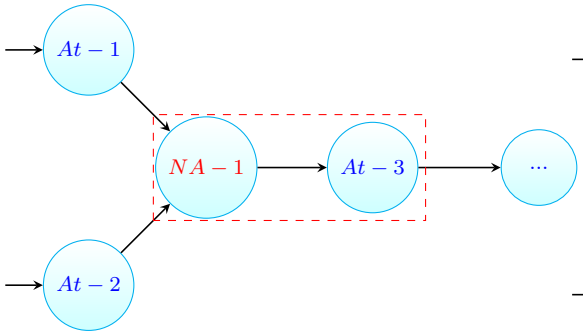


Figure 3.16: Multiple links to a non-atom vertex.

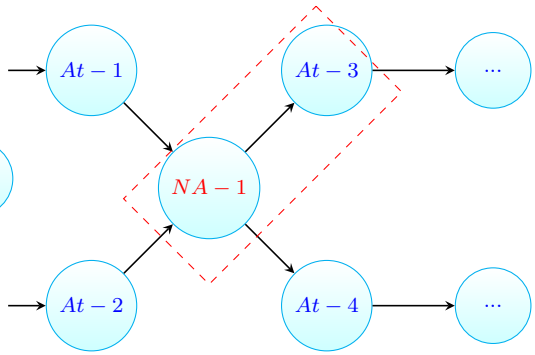


Figure 3.17: Multiple links from/to a non-atom vertex.

Each vertex of the irredundant graph is examined. The non-atom location is assigned as a priority to an atom vertex on its left side. In the illustration shown in Figure 3.14, a non-atom NA is assigned to the same location as $At-1$. In the case of Figure 3.15, there is a sequence of non-atom vertices. $NA-1$ is assigned to the same location as $At-1$ and $NA-2$ is assigned to the same location as $NA-1$. In the case of Figure 3.16, there are many branches connecting to a non-atom vertex, but there is only one link on its right side. Thus, the location of the non-atom is determined based on the location of its right side's vertex. In the case of Figure 3.17, many branches connect to the non-atom vertex from both its left and right sides. Its location is assigned according to the first branch on its right side $At-3$.

In general, condition terms are used to evaluate already linked variables. Thus, they have to be co-located with an atom vertex as soon as all its variables are assigned values.

After locating all the vertices, we label them according to the direction of the edges, in order to calculate the needed volume of communication. If the next vertex has the same location as the current one, the vertex labeling is ignored (the edge is presented by a dash line) as shown in Figure 3.18 and Figure 3.19.

Messages between two locations are generated by local rules and include all data to be transferred. For example in Figure 3.20, where two sets of data have to be transferred from S1 to S2 ((m; t) from B1 to H, and (m; k) from B2 to H respectively), only one message is transmitted, including the variables (m; t; k)

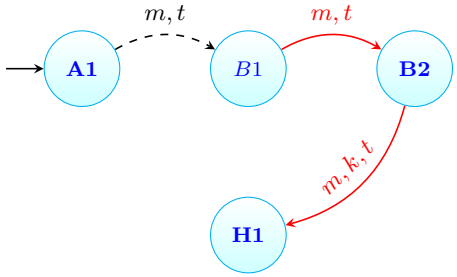


Figure 3.18: Labeled graph of the first acyclic irredundant graph from Figure 3.8.

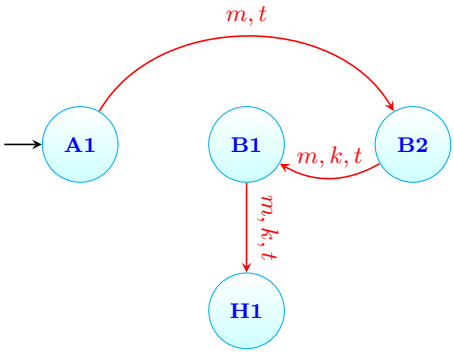


Figure 3.19: Labeled graph of the second acyclic irredundant graph from Figure 3.9.

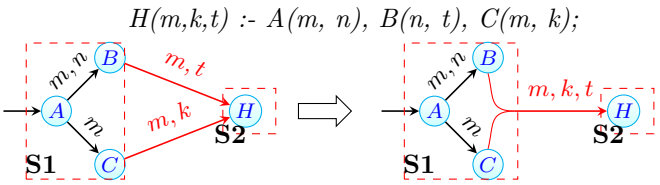


Figure 3.20: Simplify connections between two locations in a rule.

3.3.2.5 Process synchronization

It is always possible to build a minimal acyclic directed graph for a global rule comprising only one input (the first term of the body) and only one output (the head of the rule), and to partition according to the data locations. Each location evaluate the local rules that are supposed to be independent. These cooperating locations have thus to be synchronized to achieve equivalent evaluations as the original global rule. The synchronize locations ensure that messages contain sufficient information to match data included in messages coming from different locations.

For example, in Figure 3.21, S_5 receives two messages from S_2 and S_4 (m_2 and m_5 respectively). These two messages must be synchronized, so we need to find the location at the origin of m_2 and m_5 (here S_1). More precisely, we need to identify the term in S_1 at the origin of these two messages. m_2 is a consequence of m_1, generated from terms A, B, C and D. m_5 is a consequence of m_4 and m_3, the latest one is generated from terms A and B. The term causing the need of synchronization is thus B. So the key of B has to be propagated in all outgoing messages from location S_1 until location S_5.

The communication cost of each candidate minimum acyclic graph must now be computed in order to select the most efficient one.

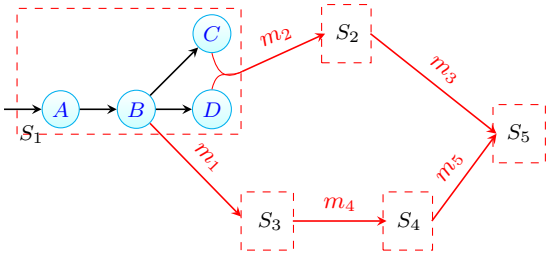


Figure 3.21: An illustrated graph for synchronization process.

3.3.2.6 Decision of communication

This part is critical to optimize the distributed computing process. All the labeled graphs are examined to choose a graph that allows taking the best decision regarding the communication criteria. With the hardware architecture hypotheses of [67], the communication decision is evaluated as an objective function with priority parameters in a descending order defined as follow:

1. Minimize the number of communications;
2. Minimize the number of transferred data-items.

In fact, to determine precisely the influence of each criterion on the objective function, we must know the characteristics of the considered network such as bandwidth, error rate, communication cost, etc. In the framework of this thesis, we do not approach the cost model of communications in the context of smart grids, which represents a perspective work.

From the point of view of the distributed programming, to simplify, we formulate the objective function $\mathcal{F}$ as a nonlinear function of two criteria.

$$minimize \quad \mathcal{F} = x * y \tag{3.21}$$

Where x is number of communications per rule, and y the total number of data-items transferred per rule. The decision of communication respects the optimal labeled graph which possesses the minimum of the objective function (3.21) .

The optimal labeled graph allows determining the optimal rule execution order in which the behavior of a rule is still transformed in the same way as in the distributed system. As mentioned above, the transferred data items must be started from the root of the labeled

graph (the vertex holds the first term of the body part) and end at the destination, which holds the head part of the rule.

Let us call the objective function of the two labeled graphs (presented in Figure 3.8 and Figure 3.9) $\mathcal{F}_1$ and $\mathcal{F}_2$, respectively. We consider that the optimal labeled graph represents the minimum of its objective function. For example, in rule (3.5), the optimal labeled graph is the graph of Figure 3.18, because in Figure 3.19, there are 3 communications and 8 transferred data-items, but in Figure 3.18 there are only 2 communications and 5 transferred data-items, thus $\mathcal{F}_1 = 2 * 5 < \mathcal{F}_2 = 3 * 8$.

3.3.2.7 Rule generation

The decision of communication complies with the optimal graph. The direction of the communication between two labeled vertices respects the direction of the edge, and the transferred data items as the edge's weight. Data is packed in the form of a temporary `data_type` and sent to another location by message passing. The name of the temporary `data_type`, which is generated automatically, allows triggering the next actions in other locations.

The name of the temporary `data_type` is chosen so that temporary `data_types` with different attributes cannot have the same name. In the tool implementation, we name the temporary `data_type` in the form "`Tmp + R + index-1 + NameModule + index-2 + G + index-3`", with `index-1` being the order of the original rule in the module, `index-2` its order in the list of rewritten rules, and `index-3` its label order in the graph. For example, in the optimal graph of Figure 3.18, a temporary `data_type` named `TmpR1A1G1` presents data-items transferred to the first label $B2$ in the first rewritten rule of rule (3.3) in the module A.

Temporary `data_types` are declared to the `data_types` block in both locations (sender and receiver). The data type `IPMap` is added at the end of the body part of the rule to indicate the IP mapping of the target address as well. For example, the data sent for the optimal graph of Figure 3.18 is expressed by sub-rules in Listing 3.8 in the site $S1$ and in Listing 3.9 in the site $S2$. Once the data instance of `data_type A1` changes, the rule in Listing 3.8 reacts and the result of this rule is sent to the address of site $S2$. After

receiving the data-items of `TmpR1A1G1`, the rule in Listing 3.9 of the site $S2$ is triggered, the result of rule execution is an update of the data items in `H1` in the site $S3$.

```
Module(A1){
^TmpR1A1G1(x, t) :- A1(x, y, t), B1(x,_), IPMap('S2', @ip); }
```

Listing 3.8: Sub-rules generated in the site $S1$.

```
Module(TmpR1A1G1){
^H1(x,k,t) :- TmpR1A1G1(x, t), B2(x, k), IPMap('S3', @ip); }
```

Listing 3.9: Sub-rules generated in the site $S2$.

After transforming the original rule into sub-rules in which all terms of the body part have the same location, each sub-rule is assigned to its location.

3.3.3 Program generation

After the rule distribution, the next step is to generate *distributed Smartlog* programs corresponding to each participating computing units. Generating a distributed program is the same as writing a *Smartlog* program with three main blocks: `data_types`, `initial_data` , and `Modules`.

3.3.3.1 Generating the `data_types` block

The `data_type` block in an individual program that can be reconstructed from the fragmentation of the original `data_types` block and temporary `data_types` which are generated in the rule generation process. The temporary `data_types` are assigned to their locations during the rule processing.

In this section, we present how to generate `data_types` blocks for each location. Based on the description of the data distribution, each `data_type` declaration (D_i) in the `Data_type` block (D) is approved one by one. The original `data_type` is rewritten to new `data_types` (D_{ij}) according to its fragments (F_i). Algorithm 2 presents the distribution of the `data_type` block. In the case of a horizontal fragmentation, the fragment has no attribute declaration part (A_{ij} is NULL). The new `data_type` form is the same as its

Algorithm 2 `Data_types` distribution.

```
1: procedure DISTRIBUTEDATATYPES(data_type_bloc)
2:     for D_i in D do
3:         for F_ij in F_i do get A_ij from F_ij
4:             if A_ij is null then
5:                 Rewrite new data-type D_ij with A_i
6:             else
7:                 Rewrite new data-type D_ij with A_ij
8:             end if
9:             Assign new data-type D_ij to L_ij
10:        end for
11:    end for
12: end procedure
```

original `data_type`. For example, the original data type A, described in Listing 3.3 is rewritten separately in two local databases, $S1$ in Listing 3.10 and $S2$ in Listing 3.11.

```
A1(Value1 : int key, Value2 : int, Value3 : int).
```

Listing 3.10: New data_type $A1$ in site $S1$.

```
A2(Value1 : int key, Value2 : int, Value3 : int).
```

Listing 3.11: New data_type $A2$ in site $S2$.

3.3.3.2 Generating the `Initial_data` block

All `initial_data` (I_i) in the `initial_data` block ($I_i \subset I$) are rewritten according to the `data_type` (D_i) and distributed to their own locations. Each fragment (F_{ij}) of the `data_types` (D_i) is evaluated based on its *condition* (C_{ij}) before assigning a new `initial_data` (I_{ij}) to its location (L_{ij}). This process is shown in Algorithm 3.

Algorithm 3 `initial_data` distribution.

```
1: procedure DISTRIBUTEINITIALDATA(initial_data_bloc)
2:     for I_i in I do
3:         for F_ij in F_i|F_i ∈ D_i do get C_ij from F_ij
4:             if C_ij is null then
5:                 Rewrite new initial data I_ij
6:             else if I_i is satisfied C_ij then
7:                 Rewrite new initial data I_ij
8:             end if
9:             Assign I_ij to L_ij
10:        end for
11:    end for
12: end procedure
```

Using the description of the `data_type` A from Listing 3.3, an `initial_data` block is described using Listing 3.12.

```
Initial_data{
A(1, 2, 3).
A(3, 2, 4). }
```

Listing 3.12: A sample "initial_data" block.

The initial data distribution in two sites $S1$ and $S2$ is proposed in Listing 3.13 and Listing 3.14 respectively.

```
Initial_data{
A1(3, 2, 4). }
```

Listing 3.13: In site $S1$.

```
Initial_data{
A2(1, 2, 3).}
```

Listing 3.14: In site $S2$.

3.3.3.3 Modules

A *Smartlog* program is generated for each location by a set of its own `data_types`, `initial_data` and rules that is grouped in modules according to their first atom in the body part.

The individual analysis of each rule in the decision of communication sometimes produces rules duplication and replicates the message transfer through two machines. In fact, the activation of two duplicated rules in the same module produces the same update of the head part. That leads to unnecessary duplicate computations and may have unfortunate side effects. A simplification of the program should be considered to eliminate this kind of redundancy as well, as a prospective research.

3.4 Validation of the CPDE methodology

We have built a tool named semi-automatic rule distribution (SARD) implementing the CPDE method. The SARD tool takes a *centralized Smartlog* program and the description of the data distribution, and generates a set of cooperating *distributed Smartlog* programs. These programs must be installed in the distributed system in order to perform distributed computations. The objective of the validation process is to verify that the behavior of the distributed execution conforms to the initial centralized one.

3.4.1 Validation methodology

The CPDE is validated by considering and evaluating behaviors of the SARD tool. Test cases using the SARD tool are proposed as incremental tests in which the complexity is ascending, and each test leans to the result of the previous ones. The test is constituted of a *centralized Smartlog* program, a description of the data distribution and a set of data-items insertion/modification which allows observing its behaviors (test scenarios). The *centralized Smartlog* program, after being analyzed by the SARD, is installed in the distributed system. Data items are inserted/updated to compare their behaviors in the two versions of the implementation (the centralized one and the distributed one). If the result of these two versions is identical, then the validation of the CPDE is positive.

3.4.2 Test campaign

The incremental tests with the structure of the test plan are presented below:

- Centralized test:

  - ◇ Rules containing only atoms;
  - ◇ All co-located atoms;
  - ◇ No fragmentation (1 fragment per `data_type`).
- Fragmentation tests:
  - ◇ Rules containing only atoms;
  - ◇ All co-located atoms;
  - ◇ Horizontal fragmentation;
  - ◇ Vertical fragmentation;
  - ◇ Hybrid fragmentation.
- Distribution tests:
  - ◇ Rule containing only atoms;
  - ◇ Distributed body part;
  - ◇ Co-located body part but separated head part.
- Distribution fragmentation tests:
  - ◇ Combination of the two previous tests.
- Assignment tests without aggregate function:
  - ◇ With/without distribution;
  - ◇ With/without fragmentation.
- Assignment tests with aggregate function:
  - ◇ With/without distribution;
  - ◇ With/without fragmentation.
- Condition tests:
  - ◇ With/without distribution;
  - ◇ With/without fragmentation;
  - ◇ With/without assignment;
  - ◇ With/without aggregate function.

A full detailed list of the test programs and their data distribution is available in appendix B.

3.4.3 Results

The validationare used to refine/correct the CPDE methodology and its SARD implementation throughout the work. Appendix B shows the distributed programs obtained with the proposed test scenarios. We have also observed the same behaviors between the centralized and distributed versions of the test programs. The validation of the CPDE was thus approved.

3.5 Conclusion

The *Smartlog* language is developed for programming and distributed data management with a high abstraction level. Even if it is designed to be well adapted to the needs of smart grids, distributed programming is still challenging. In this chapter, we proposed a methodology, called Centralized Programming and Distributed Execution (CPDE), for greatly facilitating distributed programming in *Smartlog*.

The CPDE methodology is proposed to translate a *centralized Smartlog* programs into a set of cooperating *distributed Smartlog* programs according to a description of the distribution of data while minimizing telecommunication costs. A tool, named Semi-Automatic Rule Distribution (SARD), implementing the CPDE has been developed and thoroughly validated using incremental tests.

Chapter 4

Smart grids applications in *Smartlog*

4.1 Introduction

In Chapter 2, we developed a declarative distributed data manipulation language dedicated to smart grids applications, called *Smartlog*. Whereby, the *Smartlog* language offers to perform computations in local databases and to communicate over the network. In Chapter 3, we proposed a new methodology facilitating the distributed programming, called Centralized Programming and Distributed Execution (CPDE). It relies on automatic *Smartlog* rule distribution according to a given data distribution.

This chapter aims at experimenting the proposed language and programming methodology over four real-life smart grid applications, to illustrate the pro and cons of our proposals. The first section presents our experimental environment, involving real-time simulation and hardware in the loop (with Raspberry Pis). The next sections present each application, their motivation, the *Smartlog* implementation, and discuss the results. The first two applications require a parallel execution in each node with real data. The last two applications must be performed in the real-time simulation platform with the cooperation of multiple nodes in the network architecture and real-time data production.

The first application, daily load curve reconstruction, presented in Section 4.3, tends to saturate the communication network and central servers when the number of smart meters drastically increases. We show that the *Smartlog* parallel implementation supports more efficiently this scalability issue.

The second one, computation of bus admittance matrix, presented in Section 4.4, exploits the reactivity of *Smartlog* to reduce the quantity of computation when there is any change occurring in the system. Reactivity allows indeed an incremental programming approach.

The third one, presented in Section 4.5, is distributed secondary control in microgrids. It combines the advantages of both previous applications for distributed computation.

The last one, presented in Section 4.6, is voltage regulation of a distribution grid. It demonstrates the correctness of the CPDE methodology in a more complex, fully distributed environment.

4.2 Experimental environment

Model of power grid used as object to implement the control algorithms in *Smartlog* and is simulated in the Simulink/Matlab® environment. After testing in the Simulink environment, the model is loaded in a real-time simulation machine, an OPAL-RT target [23], for execution. The objective is to observe the response of the grid model and its interaction with the *Smartlog* language as close as possible from the real-time.

Each control node is equipped with a Raspberry Pi 2 model B to perform local computations as it hosts the node architecture shown in Figure 2.1. Depending on the number of controlled nodes in the power grid model, the number of Raspberry Pis is variable. This network of Raspberry Pis constitutes the distributed database network as mentioned in Chapter 2.

Measured data are collected from the simulated grid model through the OPAL-RT server and sent to each Raspberry Pi. At the same time, each Raspberry Pi can communicate with the others using a TCP/IP protocol and send back control values to the OPAL-RT server to set up actuators in the simulated grid.

In this set-up, OPAL-RT and the network of Raspberry Pis are configured in the same network. Thus, IP addresses supply enough information to identify nodes in the network. The architecture of the real-time simulation platform used for experimentation is shown in in Figure 4.1. The total communication time between OPAL-RT and the Raspberry Pis is estimated to be around 10 ms.

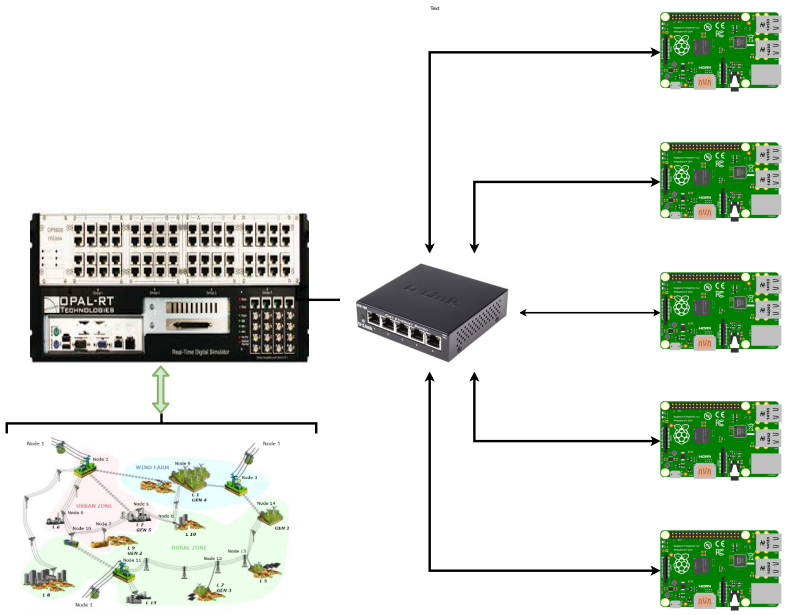


Figure 4.1: Architecture of the real-time simulation platform.

4.3 Daily load curve reconstruction

Raw data from utility data collection devices (sensors, smart meters, control devices, etc.) are sent to a concentrator and data management system at the central grid operator for processing. These devices generate a large amount of data for the central operator. That requires considerable amount of communication, processing and storage in the centralized data management system (CDMS) as shown in Figure 4.2. These requests will increase even further with the addition of new devices to the distribution grid. For centralized data management to meet these requirements, utilities must make significant infrastructure investments to improve the computing power and storage capacity of the CDMS. That leads to an increasing cost and complexity of maintaining and scaling the system, as well as more and more data latency. Thus, distributed data management systems are proposed to deal with these issues. All or part of the processing carried out by CDMS is intended to be executed in the network computing units.

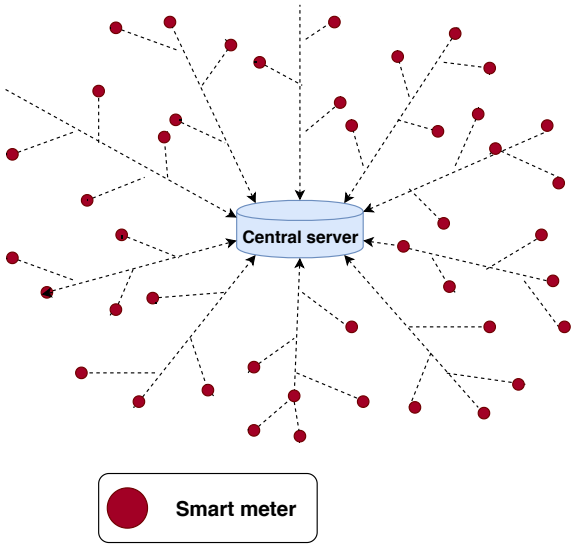


Figure 4.2: Centralized data collection.

The computing units in the network process raw data in parallel. That allows the network to naturally adapt to the addition of new computing units, and to cope with the increasing number of devices. Besides, this paradigm also helps to improve data latency [3]. Instead of a batch in the CDMS, the network computing units process data on-demand as soon as they are received. Moreover, the computing units are located near collection devices, it does not make a round trip to a central grid operator before providing information to the customers. Latency is then further minimized.

In that context, we use *Smartlog* to deploy the simple processing and aggregation of raw data in a distributed data management system (DMS). The specific application used in this section is a local aggregation of load curves. Its purpose is to illustrate the compactness of *Smartlog* and the scalability of the DMS.

4.3.1 Methodology

We assume that there are associated computing units in all smart meters (a limited computing power is sufficient), which can be in charge of receiving data measured from the smart meter and acting as a node in the architecture defined in Subsection 2.2.1.

For control services in an advanced distribution automation (ADA) [83], the average active power (POW) and reactive power (REP) is collected every 15 minutes [77]. However, for a daily monitoring application, hourly data storage is sufficient. The average of

the power consumption of the load is computed every hour, and these values are stored for 24 hours. After 24 hours, the data is deleted to free local memory.

The measured data of active and reactive power are accumulated during each hour, and then the mean power consumptions are computed and stored. Mean values aged of more than 24 hours are deleted.

4.3.2 *Smartlog* implementation

The above principle is implemented with the description of the `data_types` in *Smartlog* presented bellow:

`Measure(ID, POW, REP):` Instantaneous measure of POW and REP;

`CurrentHour(Time):` This `data_type` is updated every hour;

`Sum(ID, SUM-POW, SUM-REP, COUNT):` Accumulation of POW and REP, and number of measures;

`Curve(ID, Time, AVG-POW, AVG-REP):` Generated load curve.

Four *Smartlog* rules, grouped in two modules implementing the load curve computation are presented in Listing 4.1.

```
Module(Measure){
--R1-- Sum(x,p,q,1) :- Measure(x,p,q), ~Sum(x,_,_,_).
--R2-- Sum(x,sp,sq,c) :- Measure(x,p,q), Sum(x,spo,sqo,co), sp:=spo+p, sq:=sqo+q,
    c:=co+1.
}
Module(CurrentHour){
--R3-- Curve(x,h,ap,aq) :- CurrentHour(h),a:=h-24, !Sum(x,sp,sq,c), ~Curve(x,a,_,_),
    ap:=sp/c, aq:=sq/c.
--R4-- Curve(x,h,ap,aq) :- CurrentHour(h),a:=h-24, !Sum(x,sp,sq,c), !Curve(x,a,_,_),
    ap:=sp/c, aq:=sq/c.
}
```

Listing 4.1: Daily load curve reconstruction.

When measure data items arrive (`Measure` module), `Sum` is updated by adding the current values with new measured values and the counter is increased (rule `R2`). If

it is the first measure of the hour, there is no corresponding `Sum` and the counter value is set to 1 (rule `R1`). For every hour update (`CurrentHour` module), data items in `Curve` are produced with sum values and count values stored in `Sum` (rule `R3`). If data items in `Curve` exist before 24 hours, they are deleted (rule `R4`). For both rules `R3` and `R4`, deleting the `Sum` data item resets the sum computation.

This *Smartlog* program computes load curves for all meters from which it receives measures, either for only one smart meter (full parallel execution, program installed on every meter), a group of them (parallel group processing, one program for a group of meters), or even all of them (centralized processing, only one program for the whole power grid). Parallel computing is performed on independent subsets of measures, ensuring scalability.

4.3.3 Results and discussion

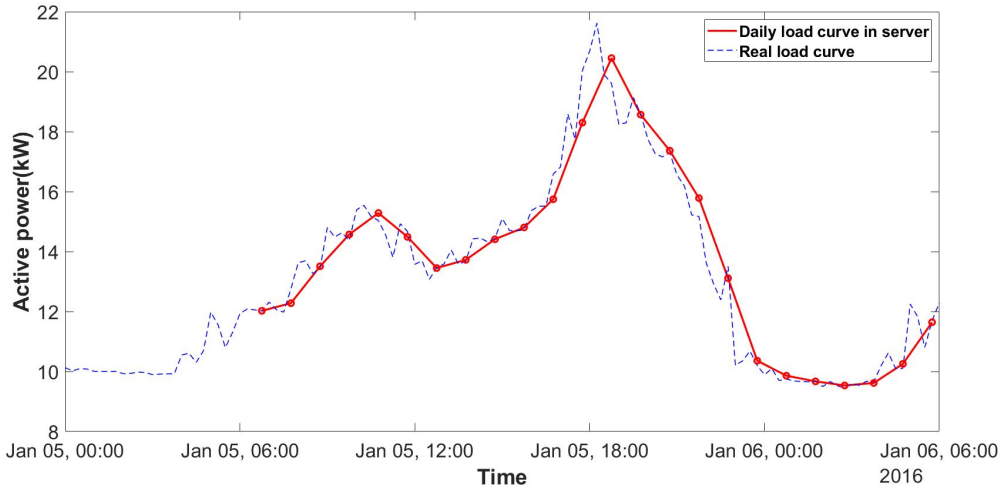
For our experiment, the data set of real measures collected every minute from a point in the distribution grid in Vietnam is used. These data are updated consecutively to the computing units. The data input includes values of active and reactive power from 0h, January 5^{th} 2017 to 6h, January 6^{th} 2017. The load curve is drawn from computing units after updating all datasets and is shown in Figure 4.3a and Figure 4.3b.

As shown in Figure 4.3, the *Smartlog* program produces the expected result. A lot of measured data are inserted to the local database, but the values stored in `data_type curve` cover only the last 24 hours. Each value in `curve` is an average value of the measured data of the previous hour.

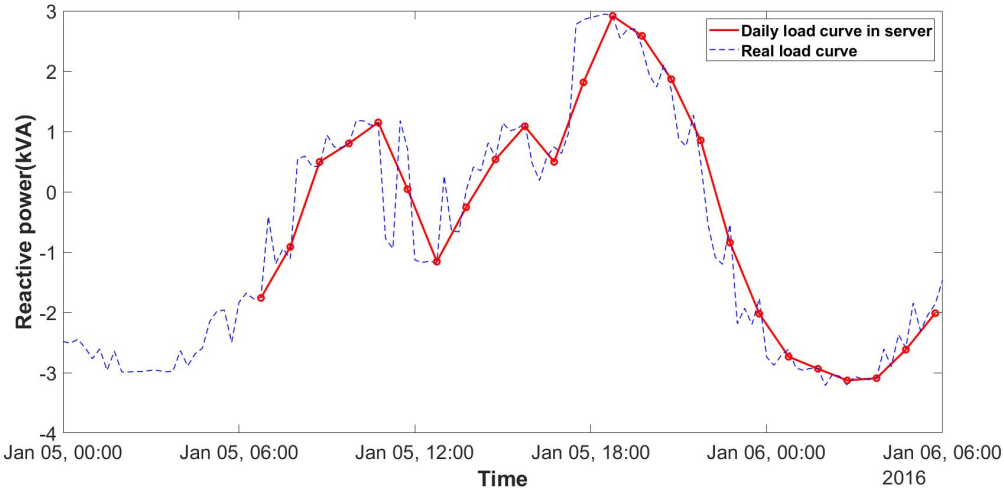
This experiment shows that the *Smartlog* language provides a simple and straightforward program that operates in parallel in each node in the DMS for which there is no need of a central server. With this first application, we can also see its compactness: The program fulfills a specific description with only four rules.

4.4 Computation of the bus admittance matrix

The bus admittance matrix ($Ybus$) is one of the data requirements needed to run a power flow study (optimal or not). The bus admittance matrix is constructed to simplify the



(a) Daily active power curve.



(b) Daily reactive power curve

Figure 4.3: Daily load curve reconstruction.

representation of a complex grid with a matrix that is more suitable to solve power balance problems with algorithms. It is developed as a useful tool for power systems. In this application, we program the admittance matrix computation in *Smartlog* in order to demonstrate reactive and incremental properties of the *Smartlog* language.

4.4.1 Methodology

The $Ybus$ matrix of a grid with N nodes is a square matrix $Ybus_{NxN}$. It is computed from the resistance and reactance measured between each pair of nodes in the actual power grid and noted R_{ij} and X_{ij} respectively. Each element y_{ij} of the $Ybus$ matrix represents the admittance of the power line connecting node i to node j, with $y_{ij} = z_{ij}^{-1} = (R_{ij} + jX_{ij})^{-1}$. y_i is the admittance-to-ground of the node i. The mathematical

expression of the $Ybus$ matrix is expressed in (4.1) [85].

$$Y_{ij} = \begin{cases} y_i + \sum_{k=1,k\neq i}^{N} y_{ik}, & \text{if } i = j \\ -y_{ij}, & \text{if } i \neq j. \end{cases} \tag{4.1}$$

Each element of the $Ybus$ matrix is computed independently, except the elements in the diagonal, which depend on the values of the other elements in the same row. An implementation of the $Ybus$ computation with Matpower 6.0 [85] is proposed in Listing 4.2 and is used for comparison with our *Smartlog* implementation.

```
1  %  Bus Admittance Matrix
2  %  Copyright (c)  1998 by H. Saadat.
3
4  function[Ybus] = ybus(zdata)
5  nl=zdata(:,1); nr=zdata(:,2); R=zdata(:,3); X=zdata(:,4);
6  nbr=length(zdata(:,1)); nbus = max(max(nl), max(nr));
7  Z = R + j*X;                              %branch impedance
8  y= ones(nbr,1)./Z;                       %branch admittance
9  Ybus=zeros(nbus,nbus);          % initialize Ybus to zero
10 for n = 1:nbus       % formation of the diagonal elements
11     for k = 1:nbr
12     if nl(k) == n | nr(k) == n
13     Ybus(n,n) = Ybus(n,n) + y(k);
14     else, end
15     end
16 end
17 for k = 1:nbr;   % formation of the off diagonal elements
18     if nl(k) > 0 & nr(k) > 0
19     Ybus(nl(k),nr(k)) = Ybus(nl(k),nr(k)) - y(k);
20     Ybus(nr(k),nl(k)) = Ybus(nl(k),nr(k));
21     end
22 end
```

Listing 4.2: Reference program in Matpower 6.0.

In the reference program, the `zdata` structure (line 5) is the input of the $Ybus$ function and is used to store the resistance (R) and the reactance (X) values. From the parameters of the `zdata` structure, the impedance (Z) and the admittance (Y) of the branches are computed (lines 7 and 8). $Ybus$ is initialized in line 9. The elements of

the *Ybus* matrix are then computed: the diagonal elements from line 10 to 16, and the off-diagonal elements from line 17 to 22, according to (4.1).

As *Smartlog* is a reactive language, we perform an incremental computation of the *Ybus* matrix. Adding or updating a value in the `zdata` structure triggers the computation of new or updated values of the *Ybus* matrix, as sums can be computed incrementally.

4.4.2 *Smartlog* implementation

In Matpower 6.0, the `Zdata` structure contains the resistance (RES) and reactance (REA) values of the power lines connecting each pair of nodes (ID1 and ID2). `Ybus` contains the real part (RE) and the imaginary part (IM) of the elements of the bus admittance matrix.

`Zdata(ID1, ID2, RES, REA):` Parameters of electrical lines;

`Ybus(ID1, ID2, RE, IM):` RE and IM of the admittance of the line from node ID1 to node ID2;

The implementation in the *Smartlog* language for the computation of *Ybus* is presented in Listing 4.3.

```
Module(Zdata){
--R1-- Ybus(i,i,re,im) :- Zdata(i,_,r,x), Ybus(i,i,ro,imo), re:=ro+(-r/(r*r+x*x)),
    im:=imo+(x/(r*r +x*x);

--R2-- Ybus(j,j,re,im) :- Zdata(_,j,r,x), Ybus(j,j,ro,imo), re:=ro+(-r/(r*r+x*x)),
    im:=imo+(x/(r*r+ x*x));

--R3-- Ybus(i,i,re,im) :- Zdata(i,_,r,x),~Ybus(i,i,_,_), re:=(-r/(r*r+r*r)),
    im:=(x/(r*r+x*x));

--R4-- Ybus(j,j,re,im) :- Zdata(_,j,r,x), ~Ybus(j,j,_,_), re:=(-r/(r*r+x*x)),
    im:=(x/(r*r+x*x));

--R5-- Ybus(i,j,re,im) :- Zdata(i,j,r,x), re:=-r/(r*r+x*x)), im:=x/(r*r+x*x));

--R6-- Ybus(j,i,re,im) :- Zdata(i,j,r,x), re:=-r/(r*r + x*x)), im:=x/(r*r+ x*x)).
}
```

Listing 4.3: *Ybus* matrix computation in Smartlog.

Because the *Ybus* matrix is symmetrical through the diagonal elements, thus the update of the line parameters leads to an update of four elements in the *Ybus* matrix. In the *Smartlog* program, when a modification of the power line characteristic in `Zdata` occurs (i.e. between two preexisting nodes), the corresponding diagonal values of *Ybus* are first updated (rules `R1` and `R2`) and then the off-diagonal values are updated (rules `R5` and `R6`).

When a new node is added to the power grid, a new power line is also added, linking the new node to a preexisting one. A new data item is then inserted in the `zdata` structure, triggering the insertion of a new diagonal admittance value for the new node (either rule `R1` or `R2`), the update of the diagonal admittance value of the preexisting node (either rule `R3` or `R4`), and the update or insertion of the off-diagonal admittance values (rules `R5` and `R6`).

The behavior of rules `R5` and `R6`, insertion or update, depends on the presence of a value in the *Ybus* matrix. Either the value exists and rules `R5` and `R6` perform an update, or they perform an insertion. Thanks to the notion of `data_type` key in *Smartlog* and also to the different terminators of the rules making them able to chain their execution.

4.4.3 Results and discussion

We consider a sample grid with the same voltage level in all nodes and ignore the line capacitors as in Figure 4.4.

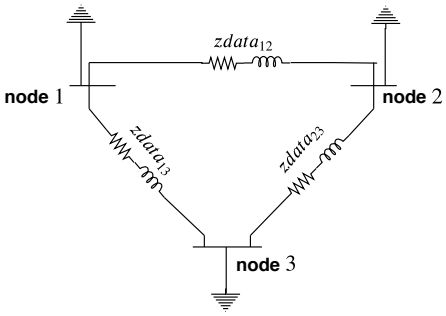
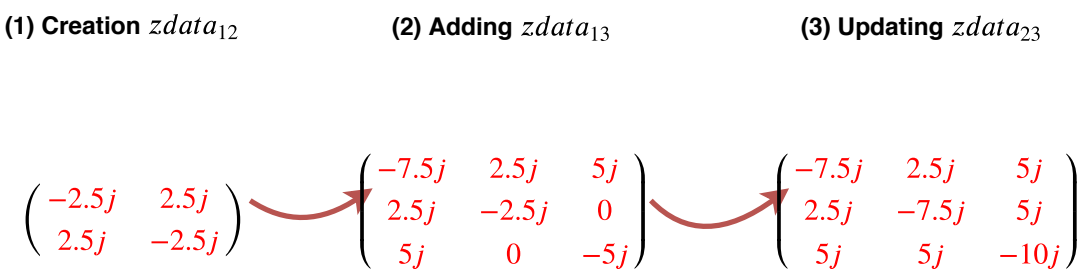
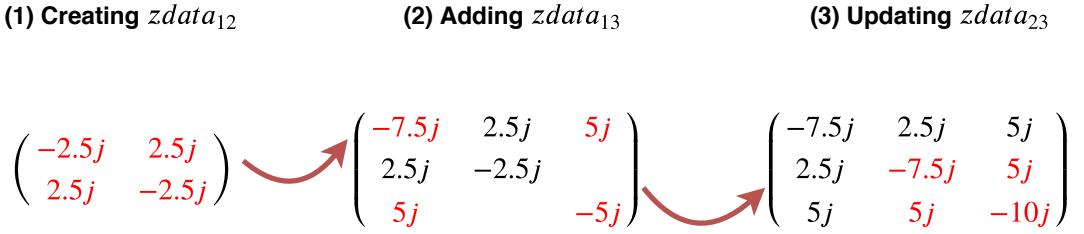


Figure 4.4: Grid support of the *Ybus* computation application.

The reference program computes the $Ybus$ matrix with the set of parameters for all the power lines as input, stored in `Zdata`. If there are any changes in `Zdata`, all parameters in `Zdata` must be reloaded and all elements of $Ybus$ are recomputed. By contrast, the *Smartlog* program "accompanies" each element in `Zdata`. By detecting a change in `Zdata`, it will immediately react to that changed parameter and will not affect the other elements of $Ybus$. As shown in Figure 4.5, describing the principle of the implementation, the size of the matrix $Ybus$ increases progressively by inserting/updating nodes (and power lines) of the grid. The first operation creates a simple power grid (one line between two nodes $N1$ and $N2$), the second one adds a new node $N3$ and a line linking it with $N1$, the last one adds a new line between preexisting nodes $N2$ and $N3$. The updated values are in red. This advantage is shown more visibly when the dimensions of the $Ybus$ matrix increases. Thus, this implementation eliminates the redundant computations. This type of programming is particularly relevant for real-time applications in smart grids in order to reduce computing time.



(a) With Matpower.



(b) With *Smartlog*.

Figure 4.5: Principle of the update of the $Ybus$ admittance matrix.

With the same set of data in $Zdata$, both programs give the same result with two different execution manner. However, *Smartlog* demonstrates its good adaptability in incremental computations.

4.5 Secondary control in an islanded microgrid

The development of microgrids is a part of the development of power systems as a whole. Power systems are potentially constructed as an aggregation of microgrids, which helps enhancing the overall resiliency and robustness of the system. The considered microgrid is designed to operate in hybrid mode, either connected to the main power grid or not (i.e. islanded mode). The microgrid contains power generator sources (e.g. renewable sources, batteries) and loads to ensure that the system can work autonomously. In normal operation, the microgrid operates in grid-connected mode, sharing local power production with the main grid. In case of main grid failure, the microgrid operates in islanded mode, maintaining its own system.

In islanded mode, all power sources must participate in maintaining the power balance. For this purpose, the primary control performed on each source will manage the balance between loads and generators. That causes possibly frequency and voltage variations. The secondary control is designed to compensate such frequency and voltage deviation in order to bring them back to nominal values.

We chose this application to focus on evaluating the implementation rather than the control method itself. Some properties of *Smartlog* such as simplicity, compactness, scalability, and incrementality are highlighted with this application. In addition, the objective is to illustrate its ability to support a distributed execution. The main points of interest are whether the execution in the local database affects the performance of the *Smartlog* implementation significantly and whether the implementation can be adapted to a near real-time operation. In this application, the computations of each node are programmed and manually distributed over the computing resources. The results are evaluated based on control criteria and compared to the implementation with an imperative language.

4.5.1 Methodology

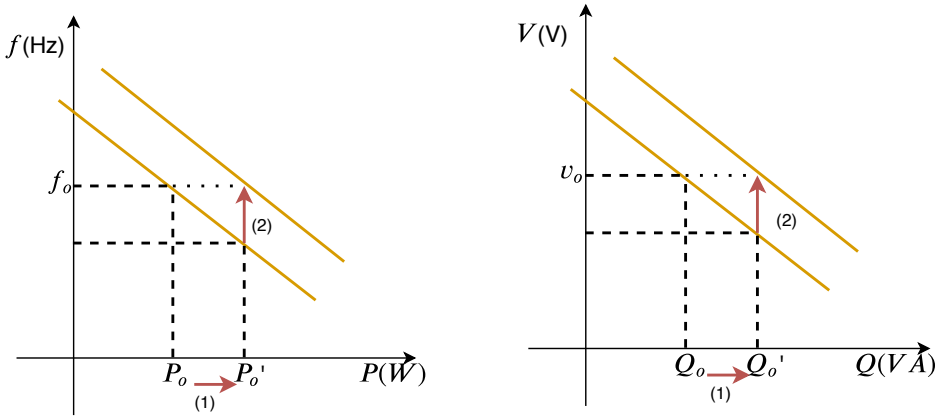
In islanded mode, the microgrid is disconnected from the main power grid. If the load varies, the primary control of each available source reacts immediately to keep the system operating in power balance (and eventually remain stable). This leads to variations in frequency and voltage magnitude, which impact the quality of the delivered energy. The

frequency and voltage deviation as a function of the power production are expressed by the droop control shown in (4.2).

$$\frac{\Delta f}{\Delta P} = \frac{f - f_o}{P - P_o} = -m \quad \text{and} \quad \frac{\Delta V}{\Delta Q} = \frac{V - V_o}{Q - Q_o} = -n \tag{4.2}$$

With f_o and V_o the set-point values of the frequency and the voltage respectively; m and n the droop gains; P_o and Q_o the initial active and reactive powers; P and Q the instantaneous active and reactive powers. The relation of f/P and v/Q in (4.2) can be represented by droop characteristic lines (DCL).

The objective of the secondary control is to bring back the frequency and the voltage at their nominal values. Ordinarily, the secondary control for power generators using alternators is taken in charge by the automatic generation control (AGC). The principle of the AGC is to move the DCL in parallel to the actual one by changing the initial power parameters of the generator [62]. As shown in Figure 4.6, increasing the initial power value (from P_0 to $P_0^{'}$) moves the DCL to the right and consequently act on the frequency. The same behavior is observed for reactive power and voltage.



(a) Frequency and active power.

(b) Voltage and reactive power.

Figure 4.6: The principle of the secondary control used in the AGC.

The resources in a microgrid scenarios are mostly DER such as photovoltaic panels and storage systems. The integration of DER, which usually contains DC sources, implies the use of a voltage source inverter (VSI) [62]. In our application, we consider a combination of three elements: (i) an array of photovoltaic panels producing a DC current, (ii) a battery facilitating the power control by storing or discharging power on demand, and (iii) a VSI converting the DC current into AC and emulating a primary control.

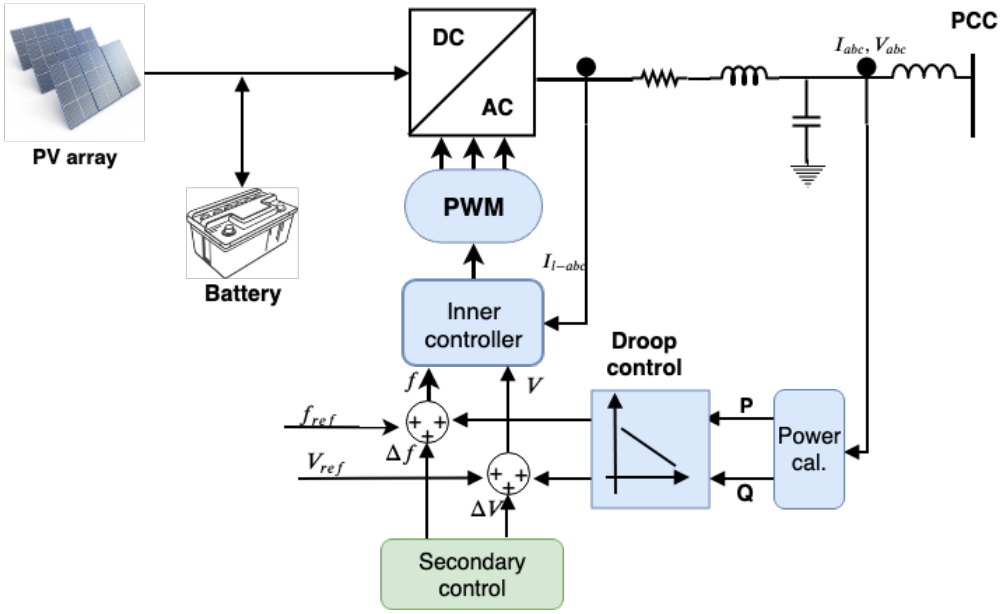


Figure 4.7: Diagram of the integration of DER into the AC microgrid.

The power calculation block instantaneously computes the active and reactive powers from measured three-phases voltage and current data. Meanwhile, the inner controller, part of the VSI, is in charge of tracking the voltage and current reference and sending the control signals to the pulse width modulation (PMW) unit.

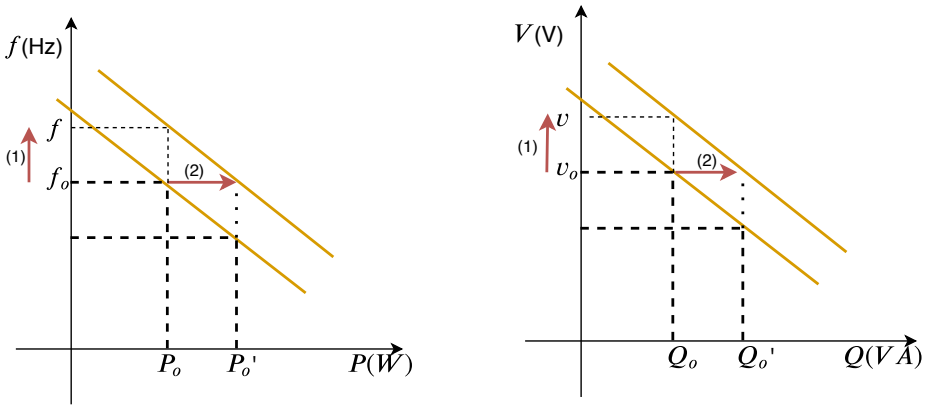
The secondary control of the VSI sends compensation values of frequency and voltage (Δf and ΔV, respectively) to the primary control to bring the frequency and voltage values back to the nominal ones. The values of Δf and ΔV are specific to each power source and must be computed using the characteristics of all sources. As shown in Figure 4.8a, increasing the frequency parameter (from f_0 to f) permits to move the DCL up and thus to increase the power while maintaining the nominal frequency.

To compute the values of Δf and ΔV, we start with:

$$\begin{cases} P'_o = P_o + \Delta P \\ Q'_o = Q_o + \Delta Q \end{cases} \quad \text{with} \quad \begin{cases} -m\Delta P = \Delta f \\ -n\Delta Q = \Delta V \end{cases} \tag{4.3}$$

The derivative of the frequency and the voltage are computed in (4.4).

$$\begin{cases} \dot{f} = \dot{f}_o - m(\dot{P}_o - \dot{P}) \\ \dot{V} = \dot{V}_o - n(\dot{Q}_o - \dot{Q}) \end{cases} \tag{4.4}$$



(a) Frequency and active power.

(b) Voltage and reactive power.

Figure 4.8: The principle of the secondary control used in the application.

In the secondary control, the set-point frequency (f_o) and set-point voltage magnitude (V_o) are constant, but the initial active and reactive powers are changed by a quantity ΔP and ΔQ respectively. So, (4.4) is rewritten in:

$$\begin{cases} \dot{f} = -m(\dot{P} + \dot{\Delta P}) \\ \dot{V} = -n(\dot{Q} + \dot{\Delta Q}) \end{cases} \tag{4.5}$$

Combining (4.5) and (4.3), we get:

$$\begin{cases} \dot{\Delta f} = \dot{f} + m\dot{P} \\ \dot{\Delta V} = \dot{V} + n\dot{Q} \end{cases} \tag{4.6}$$

To ensure an effective active power sharing, the product of the active power and the droop coefficient of each generator must be identical [35]. This is expressed in (4.7).

$$m_i.P_i = m_j.P_j \tag{4.7}$$

The *synchronous tracking error* method [13] allows synchronizing the frequency, the voltage and the active power sharing of all DER-based clusters or microgrids. It defines the derivative of the frequency, the voltage and the active power sharing for each generator i as:

$$\begin{cases} \dot{f}_i = c_f \times e_f^i \\ \dot{v}_i = c_v \times e_v^i \\ m_i.\dot{P}_i = c_p \times e_p^i \end{cases} \tag{4.8}$$

Where c_f, c_v and c_p are control coefficients. The *local neighborhood errors*, e_f^i, e_v^i and e_p^i, are defined with (4.9). Herein, the coefficients c_f, c_p and c_v are chosen identical, and from now on are called c-coefficient.

$$\begin{cases} e_f^i = \sum_{j \in N_i} a_{ij}.(f_j - f_i) + a_{i0}.(f_o - f_i) \\ e_v^i = \sum_{j \in N_i} a_{ij}.(V_j - V_i) + a_{i0}.(V_o - V_i) \\ e_p^i = \sum_{j \in N_i} a_{ij}.(m_j.P_j - m_i.P_i) \end{cases} \tag{4.9}$$

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

The derivation of the reactive power is computed with a low-pass filter, expressed in (4.10) in the Laplace domain.

$$Q = \frac{w_c}{w_c + s} Q' \Rightarrow Q.s = w_c.(Q^{'} - Q) \equiv \dot{Q} \tag{4.10}$$

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 must be transformed in their discrete-time representation. The output is then calculated as follows:

$$\begin{cases} \Delta f_i = \int \dot{\Delta f_i} dt \\ \Delta V_i = \int \dot{\Delta V_i} dt \end{cases} \tag{4.11}$$

These Laplace functions are transformed into discrete time with T_s as sample time.

$$\begin{cases} \Delta f_i(k) = T_s u_f(k) + \Delta f_i(k-1) \\ \Delta V_i(k) = T_s u_v(k) + \Delta v_i(k-1) \end{cases} \quad \text{with} \quad \begin{cases} u_f(k) = -c_f e_f^i(k) - c_p e_p^i(k) \\ u_v(k) = -c_v e_v^i(k) + u_q(k) \\ u_q = -n_i(Q(k) - Q(k-1))/T_s \end{cases} \tag{4.12}$$

The method converges if $T_s \times c < 1$ [58], c relates to the response time of the system. To evaluate the performance of the *Smartlog* implementation, the c-coefficient will be fixed, and T_s will vary, because T_s depends mostly on the sum of the computation and communication time. The value of T_s will affect 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 convergence criteria, the system will be unstable and possibly go out of control.

4.5.2 *Smartlog* implementation

In the context of this application, we denote the identification (ID) of each node as an integer number, the set-point values of frequency and voltage are 50 Hz (standard frequency in Europe) and 311 V (nominal voltage in three-phases low-voltage grid) respectively. The c-coefficient is chosen depending mainly on the network characteristic. T_s itself is always bigger than the delay time of each communication T_{delay}. Based on the condition of method convergence, the value of C is chosen so that $T_s > T_{delay}$ and at least, with a T_s guarantying that the control method converges in the traditional implementation. In our case, C is set to 2.

The description of the `data_type`s used in the *Smartlog* program is proposed as follows:

`Measure(ID, FRE, VOLT, POW, REP):` Instantaneous measure of the frequency, voltage, active and reactive powers for each node;

`Droop(ID,` m`,` n`):` Droop constants for each DER generator;

`DynamicMeasure(ID, FRE, VOLT,` $mPOW$`,` $nREP$`, REP):` Data sent to neighbor nodes;

`Neighbor(ID, ID-neighbor, Address):` Neighbor name/network address mapping;

`NeighborData(ID, ID-Neighbor,` δf_{ij}`,` e_f^i`,` δv_{ij}`,` e_v^i`,` δp_{ij}`,` e_p^i`):` Local neighbor errors of frequency, voltage and power sharing;

`NeighborSum(ID,` $\sum \delta f_{ij}$`,` $\sum \delta V_{ij}$`):` Sum of local neighbor errors;

`Output(ID,` Δf_i`,` ΔV_i`):` Compensation values of frequency and voltage for each node;

The program, shown in Listing 4.4, is loaded into each computing unit for real-time experimentation.

```
Module(Measure){
--R1-- & 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){
--R2-- 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;
--R3-- 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;
--R4-- 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){
--R5-- NeighborSum(i,ef,ev):- NeighborData(i,_,_,sdf,_,smp,_,sdv),
    NeighborSum(i,df,dv), ef:=df+sdf+smp, ev:=sdv+dv.}
```

Listing 4.4: Distributed secondary control implementation in Smartlog.

When a data item in `Measure` is modified and its frequency value violates the setpoint value, the data item in `DynamicMeasure` are sent to the neighboring nodes (rule `R1`) and the neighborhood error is updated in the current node (rule `R2`). Otherwise, if a node receives the modification of neighbor's data, it also updates the neighborhood error (rule `R3`). Every change in the neighborhood's error leads to the incremental update in the sum of the neighborhood's errors (rule `R5`). The output control values consecutively change if there is a modification of each measured data item, itself or one of its neighbors (rule `R4`).

4.5.3 Experimental results

To illustrate the behaviors of the secondary control, we chose a simplified islanded microgrid with three distributed energy resources (DER) which supply AC static loads, as shown in Figure 4.9.

The parameters of the VSI as well as the coefficients of the primary control are presented in detail in appendix A. The model is constructed and simulated in the Simulink/Matlab environment, then it is executed in the experimental environment as described in Section 4.2.

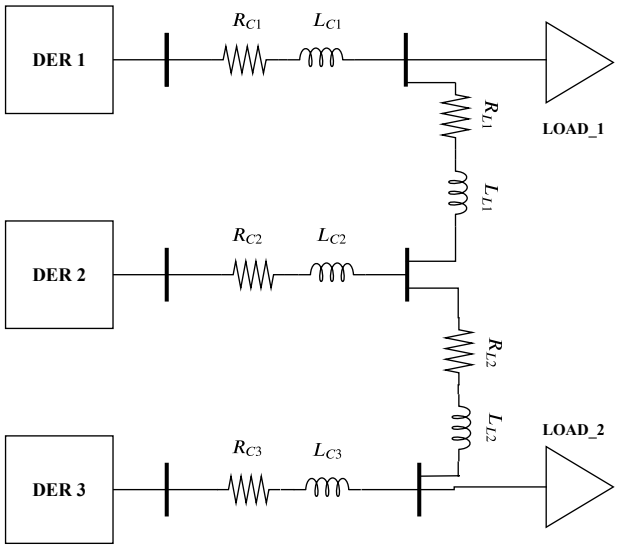
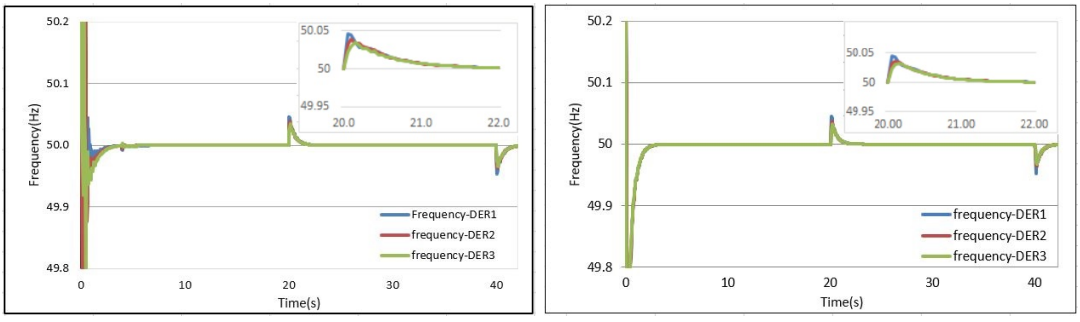


Figure 4.9: Simulated microgrid to illustrate the secondary control [75].

We perform the simulation of the microgrid in islanded mode for 60 s and load changes at T=20 s and T=40 s by a variation of 25 % of the maximal power. The parameters of the primary control are computed so that the stability of the grid, when the load changes, should be reached after 0.5 s.

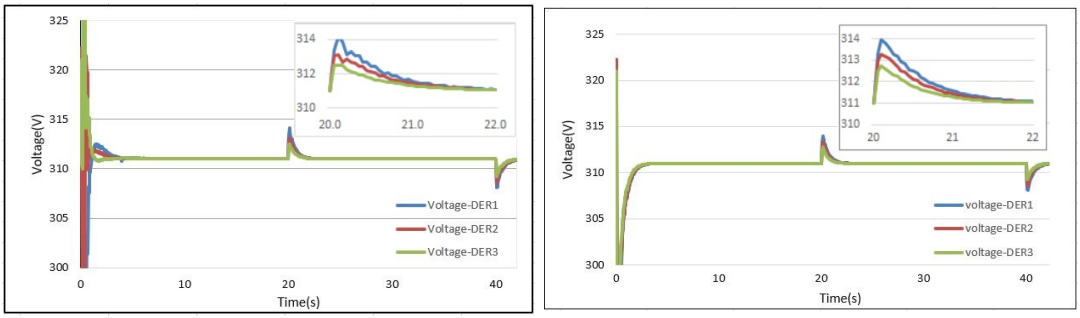
At times T=20 s and T=40 s, the load change leads to the change of frequency. The secondary control is activated to restore the frequency and voltage magnitude to their set-point value and share the powers among resources, as shown in Figure 4.10a, Figure 4.11a and Figure 4.12a.



(a) *Smartlog* implementation. **(b)** Java implementation.

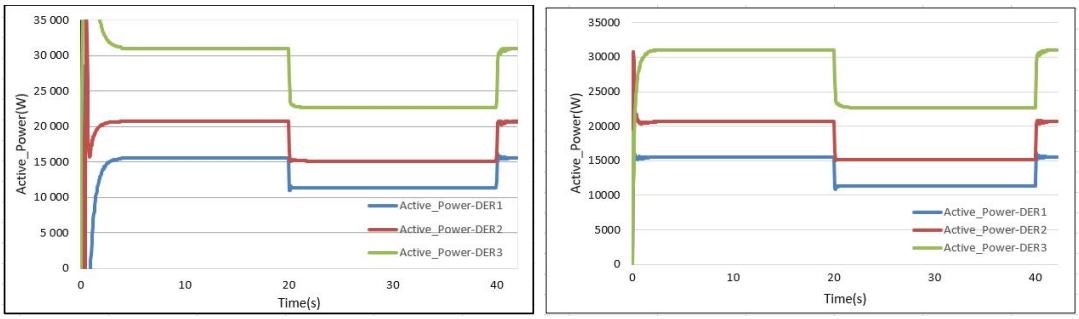
Figure 4.10: Frequency response after a 25 % load change.

We focus on considering the behaviors of the controllers facing the changes in the system. The fluctuation of the starting moment is ignored as it should only(rarely) occur in the microgrid life (i.e. transitions from connected mode to islanded mode). The



(a) *Smartlog* implementation. **(b)** Java implementation.

Figure 4.11: Voltage response after a 25 % load change.



(a) *Smartlog* implementation. **(b)** Java implementation.

Figure 4.12: Active power sharing after a 25 % load change.

incremental programming implementation in *Smartlog* in the real-time simulation platform obtains the same desired results as in the secondary control theory. These results, from distributed *Smartlog* implementation, allows once again confirming the distributed fixpoints of the *Smartlog* program as well as the distributed programming mechanism of *Smartlog*. In the following section, we discuss on qualitative evaluations in *Smartlog* and its limitations.

4.5.4 Discussion

4.5.4.1 Simplicity

The *Smartlog* language supports working simply with database management and communication. It uses simple symbols ("^", "@") to describe communications for which others languages need much more command rules. Moreover, it is not necessary to declare variables in modules because this declaration is made automatically and optimally with *Smartlog*. In each component, a *Smartlog* program clarifies *which* subproblem

in the system is activated and the corresponding commands are executed, so that no additional commands are executed.

4.5.4.2 Compactness

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 a program being more perspicuous. Indeed, the secondary control is programmed in only five rules in *Smartlog*. 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 and not only follow the steps of the algorithm. The output of the control is immediately adjusted after a change in the data.

For comparison, the same implementation in Java JDK 1.8.0 (whose data are stored in the temporary memory, not in the database) is used as a reference scenario. The Java program is available in [78]. The comparison of both implementations allow confirming the quantitative evaluations as well as the impact of an execution in the database on the response of the *Smartlog* implementation.

The results of the secondary control implemented in Java are shown with the frequency (Figure 4.10b), voltage magnitude (Figure 4.11b), and power sharing (Figure 4.12b) as a function of the time. The response-time of both implementation of the control are similar in both programming languages. However, the number of command lines in Java is much more significant than in the *Smartlog* implementation. The compactness of *Smartlog* in this test-case is summarized in Table 4.1.

Table 4.1: Number of command lines and instructions in *Smartlog* and Java for the secondary control implementation.

	Smartlog	**Java**
Lines	5	54
Instructions	36	108

The implementation in *Smartlog* needs almost 10 times less lines of code and 3 times less instructions than the implementation with Java. Looking at the level of the number of machine-level instructions is the next step of this comparison but was not easily

accessible for the experiment, because both the Java and the *Smartlog* implementations were not directly compiled into machine code.

4.5.4.3 Increment and scalability

Because *Smartlog* is designed for declarative programming, the computation is based on detecting changes in the database. *Smartlog* supports algorithms developed for real-time control based on the current measure and history data. The output values are corrected step by step. For this reason, no sample communication timing between neighbors are fixed. Those timings are defined by the interval between updates in data and the actual computation time.

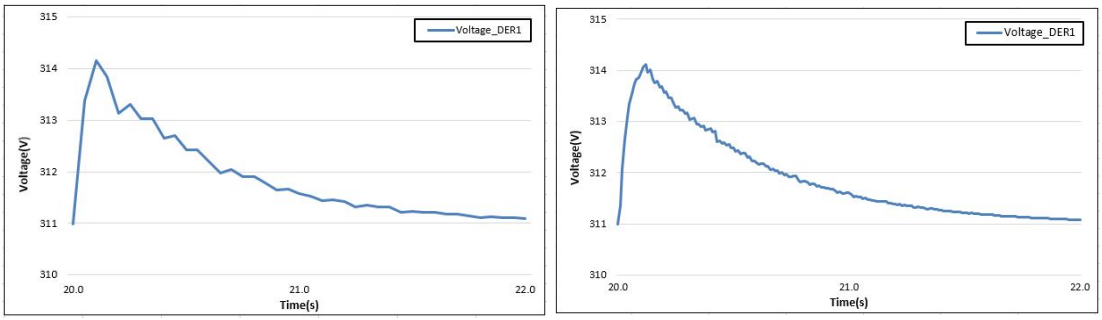
Moreover, by keeping the same configuration of network and the same program (modules), the performance of the load computation is conducted on an increasing number of Raspberry PIs. We get the same result, but the computation is shared. This proves the potential of the approach in sharing the calculations and data as the computing resource of the system grows up.

4.5.4.4 Execution in the local database

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

As stated in Subsection 4.5.1, 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) in both implementations, presented in Figure 4.13, the Java implementation is better performing. It means that T_s is bigger in the *Smartlog* implementation than in the 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 performance of *Smartlog* is apparently sufficient



(a) With *Smartlog*. **(b)** With Java.

Figure 4.13: Voltage response in both implementations.

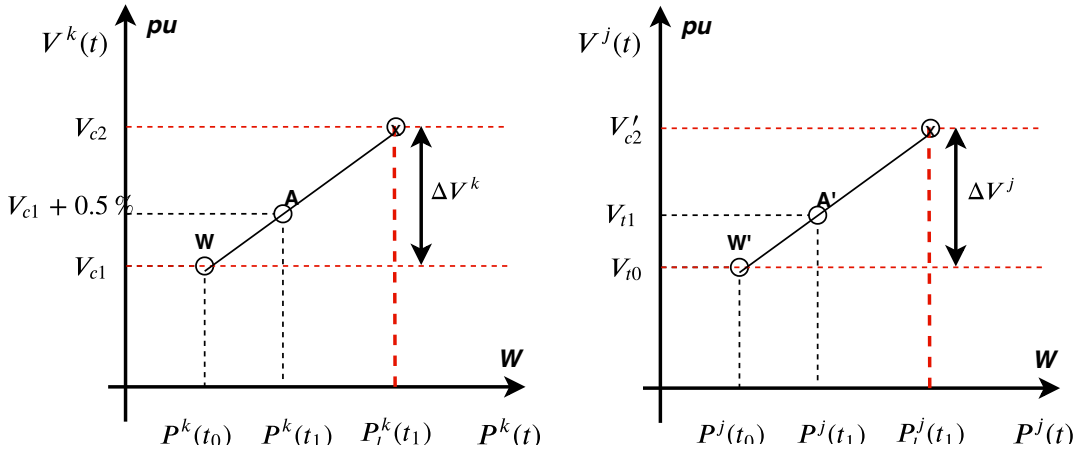
for the operating requirements in the context of this islanded microgrid, which was our initial goal. The procedure for the acceleration of the computing time in *Smartlog* is a research perspective, for example by using in-memory databases or dedicated rule execution engines.

4.6 Voltage regulation with PV production

The integration of renewable energies into power systems is a challenge for the development of smart grids. The increase in the percentage of penetration of these resources, especially photovoltaic (PV) systems into traditional power systems, causes risks of power imbalance as well as impacts the energy quality. One of the most notable issues, when this energy source is integrated into the grid, is over-voltage due to the overproduction. There are many solutions to tackle this problem, and one of the most interesting methods, being suitable for the CPDE methodology illustration, is the Adaptive Active Power Capping (AAPC) method [6]. AAPC method is preferred for execution in a central server. We approach this application as a centralized algorithm to validate our CPDE programming methodology (refer to Chapter 3). The objective is, with a well-behaved (converging) centralized program, to verify that the distributed programs also converges, and that its transformation into a set of distributed programs by CPDE does not affect the algorithm.

4.6.1 Methodology

The purpose of the AAPC method is to prevent the highest voltage node in the grid to exceed the threshold in which the node has to be cut off from the grid. In [6], there are two over-voltage thresholds: warning (V_{C_1}) and alert (cut off from the grid, V_{C_2}). They are set to $[V_{C_1}; V_{C_2}] = [1.042; 1.058]$ pu. The node having the highest voltage is called the critical node. If the voltage of a critical node is in the threshold $[V_{C_1}; V_{C_2}]$ then other nodes have also the corresponding threshold $[V_{t_0}; V'_{C_2}]$, with $V'_{C_2} < V_{C_2}$. We consider that the variation of the voltage in the distribution grid (on a short time) is assumed to be linear and proportional to the injected active power. Thus, we can predict the value of the limited active power for each photovoltaic system corresponding to its upper bound voltage at each node. If a photovoltaic system produces at its limited power, it will prevent the over-voltage. The general principle of the method is presented in Figure 4.14.



(a) Critical node **(b)** Other nodes

Figure 4.14: Principle of Active Power capping method [6].

The over-voltage occurs in one or several nodes in the grid. When the voltage of a node is superior to V_{c_1}, it is called a critical node (Cri). In the AAPC method, V'_{c_2} (Figure 4.14) is estimated based on the ratio PROV expressed as follows:

$$PROV = \frac{V_{c_2} - V^k(t_1)}{V^k(t_1) - V_{c_1}} = \frac{V'_{c_2} - V^j(t_1)}{V^j(t_1) - V^j(t_0)} \tag{4.13}$$

In principle, the AAPC uses the linear regressive method for each photovoltaic source to predict the upper power limitation (Plim) P_l^j.

$$P_l^j(t_1) = P^j(t_0) + (V'_{c2} - V^j(t_o))/\xi^j \tag{4.14}$$

with ξ the linear coefficient (Slope) calculated by the ratio of voltage variation (Vvar) and power variation (Pvar):

$$\xi^j = \frac{V^j(t_1) - V^j(t_0)}{P^j(t_1) - P^j(t_0)} \tag{4.15}$$

The photovoltaic power production within its threshold will prevent over-voltage in the grid. P^j_{ref}, the generated power of the j^{th} photovoltaic node in the next step, is its limit power (Plim) or maximal produced power (Pmax) predicted by `maximum power point tracking` calculation, defined as follows:

$$P^j_{ref}(t_1) = min(P^j_l(t_1), P^j_m(t_2)) \tag{4.16}$$

Each photovoltaic node has the same responsibility to participate in stable grid operations. Thus, the curtailment of each PV panel must be controlled. The power production of each node at time t_2, called $P^j_a(t_2)$, is expressed as:

$$P^j_a(t_2) = \frac{\sum_{j=1}^{j=n} P^j_{ref}(t_1)}{\sum_{j=1}^{j=n} P^j_m(t_2)} * P^j_m(t_2) \tag{4.17}$$

with η the power curtailment (Pcur), defined as:

$$\eta = \frac{\sum_{j=1}^{j=n} P^j_{ref}(t_1)}{\sum_{j=1}^{j=n} P^j_m(t_2)} \tag{4.18}$$

4.6.2 *Smartlog* implementation

The description of the `data_types` in this application is presented as follows:

`Measure(ID, Timestamp, VOLT, POW, Pmax):` Instantaneous measure of the voltage, current, active power and maximal predictive active power;

`Warning(ID, Timestamp):` Keeps the timestamp at which the node `ID` attains the warning status;

`WarningMeasure(WarningID, ID, VOLT, POW):` Measure the data of all nodes when the grid is in warning status;

`Alert(ID, Timestamp, PROV):` Stores the timestamp at which the critical node attains the alert status, and the corresponding value of the PROV ratio;

`Slope(Cri-ID, ID, Slope, Vvar):` Slope coefficient and Vvar for the linear regression;

`Plimit(Cri-ID, ID, Pref):` Computed injected power limit for each node;

`Curtail(ID, Pcur):` Computed power curtailment percentage of each PV node;

`Actuator(ID, Yield):` Output curtailment values for each PV node actuator;

Rules are expressed in the *Smartlog* language as follows:

```
Module(Measure) {
--R1-- Warning(i,t) :- Measure(i,t,vi,p,pmi), ~WarningMeasure(i,i,_,_), vi>=1.042;
--R2-- :- Measure(i,t,vi,p,pmi), !WarningMeasure(i,i,_,_), vi <1.042;
--R3-- Alert(i,t,co) :- Measure(i,t,vi,p,pmi),  WarningMeasure(i,i,_,_), vi >= 1.047,
    co := (1.058-vi)/(vi-1.042);
}
Module(Warning) {
--R4-- WarningMeasure (i,j,vj,pj) :-  Warning(i,t), Measure(j,t,vj,pj,_);
}
Module(Alert) {
--R5-- Slope(i,j,s,dv) :- Alert(i,t,co), Measure(j,t,vj,pj,pmj),
    WarningMeasure(i,j,vo,po), pj<>po, s:= (vj-vo) /(pj-po), s<>0, dv := (co+1)*(vj-vo);
--R6-- Curtail(i,pe) :- Alert(i,t,_), Measure(j,t, _,_, pmj), Plimit(i,j,pli), pli :=
    least(pli,pmj), prs := sum(pli), pms := sum(pmj), pe:= prs/pms;
}
Module(Slope) {
--R7-- Plimit(i,j,pli) :- Slope(i,j,s,dv), WarningMeasure(i,j,_, po), pli := po+ dv/s;
}
Module(Curtail) {
--R8-- Actuator(j,pe) :- Curtail(i,pe), Actuator(j,_,_);
}
```

Listing 4.5: *Smartlog* rules for the AAPC algorithm.

The measured data are updated regularly, if the voltage of a node in the grid reaches the warning value (1.042 pu), rule `R1` updates the warning value and the status of all nodes in the grid are saved by rule `R4`. When the voltage of the critical node reaches $V_{c1} + 0.5\,\%$

(1.047 pu), the alert value is updated by rule `R3`. The modification of the `Alert` item leads to compute the limit of the active power at each node with the AAPC method (by rule `R5, R6, R7`). The output of the control is then updated once the modification of `Curtail` is done by `R8`. If the voltage of the critical node presents a warning value, its warning status is deleted by rule `R2`.

4.6.3 Experimental result

A distribution grid from the G2Elab experimental platforms, PREDIS [5], is used as a test object for this application. This grid comprises 14 nodes, with five distributed sources, three asynchronous machines, and static loads. The grid configuration is shown in Figure 4.15. This grid is simulated in the Matlab/Simulink environment in a grid-connected mode and executed in the experimental enviroment described in Section 4.3. Each Raspberry Pi plays the role of a local computing unit which is installed near a photovoltaic source corresponding to node 5, 7, 9, 12 and 14.

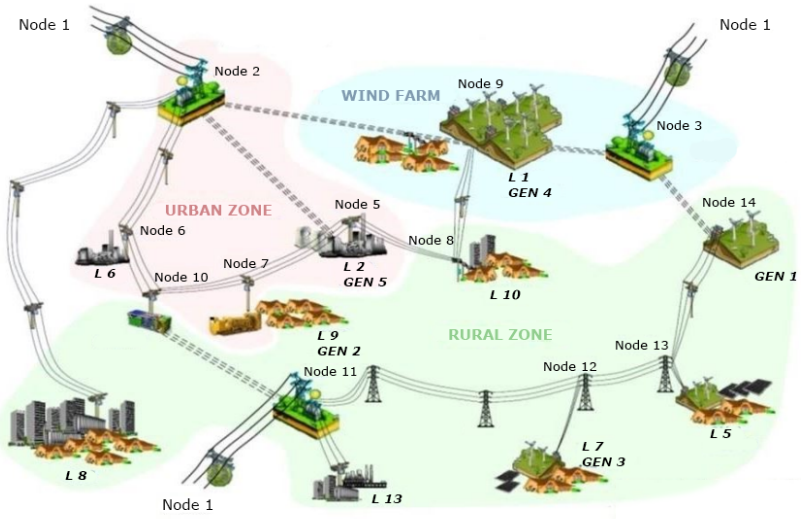


Figure 4.15: The PREDIS grid configuration [5].

In the voltage control problem, we consider the change of power injected at each node. In order to illustrate this methodology, the injected power must change by time and sometimes causes over-voltage in the grid. The experiment lasts 10 min, during which the load is kept constant and the photovoltaic production changes. The variation of solar power depends mainly on climate and clouds which cannot be precisely predicted

by time series. The curve presented in Figure 4.16 is used to evaluate the method as well as to consider the performance of the execution.

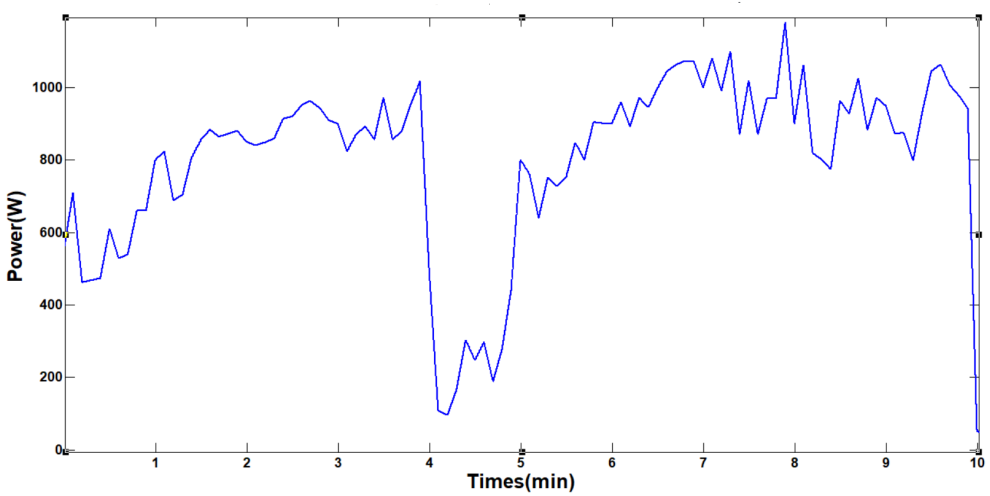


Figure 4.16: PV production of the study case.

Dataset are automatically generated during the real-time simulation. Each distributed photovoltaic source has a Raspberry Pi in charge of the local computation. There are five Raspberry Pis used in this case at nodes 5, 7, 9, 12, and 14.

According to the objective of the work, we focus on demonstrating the correctness of the CPDE methodology. We also implement the centralized algorithm in Java as a reference implementation. The result of this implementation is compared to the results obtained with the distributed *Smartlog* implementation.

We first present the data distribution of the implementation, used as input in the SARD tool. There are 8 `data_types` used in the centralized *Smartlog* program, including three common `data_types` (`Warning`, `Alert`, `Curtail`). We have five Raspberry Pis corresponding to five photovoltaic generators. Common data are only located in one machine and the other `data_types` are fragmented horizontally over the five machines. The five distributed smartlog programs corresponding to five Raspberry Pis and generated by the SARD (Section 3.4) are presented in Appendix C.

We consider the Predis model with the described scenario, the 7^{th} node, corresponding to the Raspberry Pi N^o7, is a critical node (the node with the highest voltage in the grid). Therefore, we observe the behaviors of this node with both a Java and a *Smartlog* implementations. Results are shown in Figure 4.17 for both the centralized and the distributed implementations.

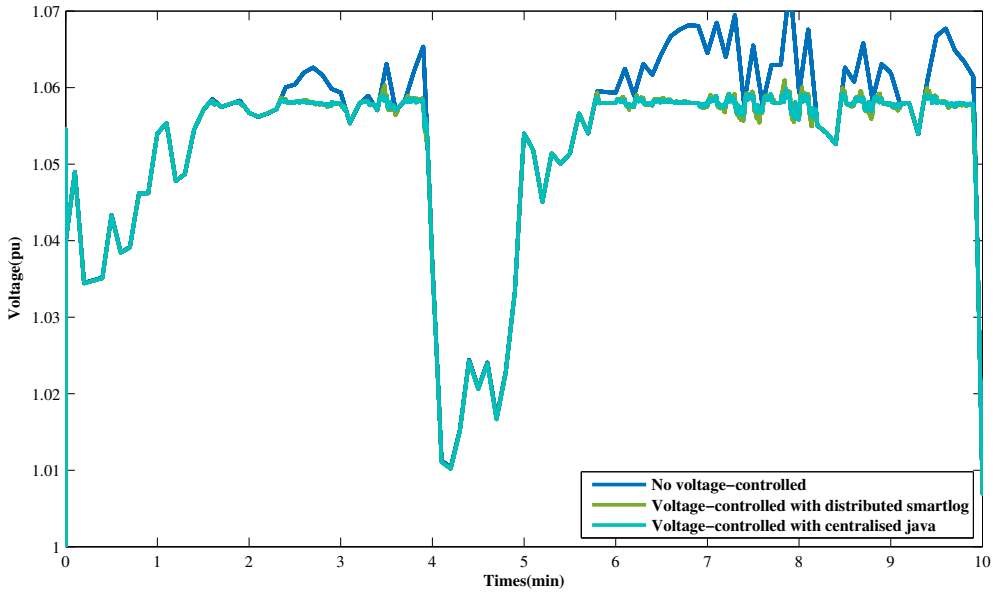


Figure 4.17: Voltage response at the 7^{th} photovoltaic node.

In general, the voltage in both implementations is controlled and does not exceed the upper bound (1.058 pu) when the grid has an over-voltage tendency.

4.6.4 Discussion

The result proves that our methodology works well in the real-time simulation platform. The centralized *Smartlog* program is transformed by the SARD according to the data distribution, and executed correctly in a distributed manner. Starting from the pre-allocation of the data distribution under the constraint of data production locations, the CPDE methodology produces a number of distributed programs. These programs cooperate in the distributed system but do not affect the centralized algorithm. Besides, this provides a simple and efficient methodology for distributed computing programming. This approach offers a simple solution for running smart grid centralized applications in distributed environments.

However, the CPDE methodology should not be stopped by confirming its correctness. It must be analyzed more on other aspects such as performance and scalability for instance. These evaluations are addressed in the next chapter.

4.7 Conclusion

The principal purpose of this chapter is to present the qualitative evaluations of *Smartlog* for applications in smart grids. By presenting applications with increasing complexity, the main benefits of *Smartlog* are highlighted, such as simplicity, compactness, and scalability, especially as the support of incremental programming. That is necessary to limit the redundancy of computations and communications for real-time applications. Besides, in this chapter, the correctness of the distributed programming methodology is also verified in real-time.

This promises a simple and easy implementation to deal with the convergence proof of distributed programs as well as the scalability of the distributed system, even when the distribution of computing resources is uneven.

The results of the test applications in the real-time simulation platform also show that the operation on database management systems more or less affects the performance of *Smartlog*. Nevertheless, its performances remain acceptable for real-time applications and are comparable to more traditional implementations. The issue of computing time acceleration needs to be tackled in the future.

Chapter 5

Experimental evaluation of a distributed *Smartlog* execution

5.1 Introduction

Distributed programming with a declarative language (*Smartlog*) is a new approach to the distributed paradigm for smart grids. In order to convince that this approach is suitable for smart grids applications, besides the qualitative evaluations mentioned in Chapter 4 as well as the correctness of the CPDE method for a distributed implementation, the effects of the SARD tool on the performance of the distributed executions should be considered.

In this chapter, we analyze three main aspects of the distributed execution in a smart grid's application. First, the performance of the distributed programming implementation is compared to the centralized deployment (traditional one). Second, the influence of the number of computing units is assessed for the case where the system scales up (which represents one of the major supposed advantage of the language). Ultimately, the impact of various data distributions on the performance is discussed. Based on the results of the implementations in the experimental environment, we point out the advantages and disadvantages of our proposed CPDE approach.

For the experiment, we continue using the application of the fair over-voltage regulation in a distribution grid with high penetration of photovoltaic sources, presented in Section 4.6. The parameters and the model of the experiment are retained for the following

evaluations. The results are observed as well in the 7^{th} node of the PREDIS grid, the highest voltage node in the scenario mentioned in previous chapter. Detail of evaluations are presented as sections below.

5.2 Performance of the distributed programming

In Figure 5.1, the voltage of the 7^{th} node is shown with a *Smartlog* implementations (centralized and distributed) and a centralized Java implementation as reference. The results are similar, which confirms the correctness of the CPDE method as well as the SARD tool. The purpose of this section is to assess the performance of the distributed execution with the SARD tool in comparison to the centralized implementations.

5.2.1 Methodology

In the case of the distributed programming, data are distributed to five Raspberry Pis. We assume that two common data `Warning` and `Alert` are located in the Raspberry Pi N^{o}7 and `Curtail` is located in the Raspberry Pi N^{o}9, the other `data_types` are fragmented horizontally over five Raspberry Pis.

We use the response time of the PV system when the system is in alert status as a criterion to evaluate and compare the performance of each implementation. As seen in Figure 5.1, during the simulation time, the voltage is controlled many times in order not to exceed its limit.

At the second minute of the simulation, the active power of the PV system increases and causes an over-voltage in the PREDIS grid. The adaptive active power capping method (AAPC) [6] is activated to restrain the percentage of the power production of each PV system. In continuous, when the node is in alert status, the power curtailment of each node is computed, which is inversely proportional to the rise of voltage. This also means that the curtailment decreases linearly when the power production increases and exceeds the power upper bound.

Practically speaking, the response time is defined as the interval between two consecutive reactions of the PV system when the grid operates in over-voltage. We thus estimate the response time of each deployment in the experiment based on this definition. Before

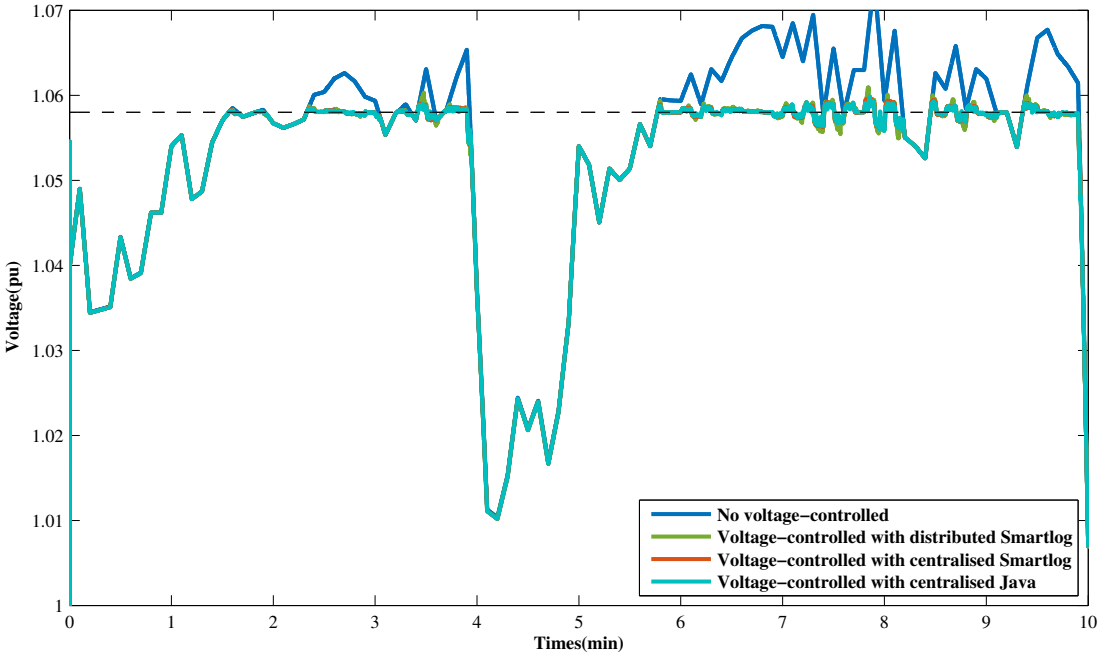


Figure 5.1: Voltage response at the 7^{th} photovoltaic node with distributed programming and centralized programming, implemented in Java and Smartlog.

showing the results of the response time estimation in the experimentation, it is analyzed according to the computation and communication times.

We call t_i the processing time of the i^{th} rule in the local node, t_c the average delay time for each communication and N the number of considered computing units in the grid. The response time of the j^{th} node is estimated with (5.1).

$$T_{res} = T_{comp} + T_{comm} = \sum_{i \in R_j} t_i + k_j * t_c \tag{5.1}$$

Where R_j and k_j are respectively the number of executed rules and the number of communications from the j^{th} node. The analysis of the response time is presented in Table 5.1 for both the centralized and the distributed *Smartlog* implementations.

Table 5.1: Analysis of the response time in both the centralized and distributed *Smartlog* implementations.

	T_{comp}	T_{comm}
Centralized programming	$\sum_{j \in N} \sum_{i \in R_j} t_i$	0
Distributed programming	$\max(\sum_{i \in R_{No7}} t_i, \sum_{j \in R_{No9}} t_j)$	Nt_c

Although the computation load is shared over the participating computing devices, the response time still relies on the communication time that depends on the characteristics

of the network. If the communication time is significant, the response time with a distributed programming may be bigger than the one of a centralized implementation.

5.2.2 Results and discussion

In the experiment, the statistics of the response time are recorded and drawn. The average response time in the three implementations is shown in Figure 5.2 with a "X" mark. The deviation from the mean of the distributed programming implementation is smaller than the one of the centralized implementations.

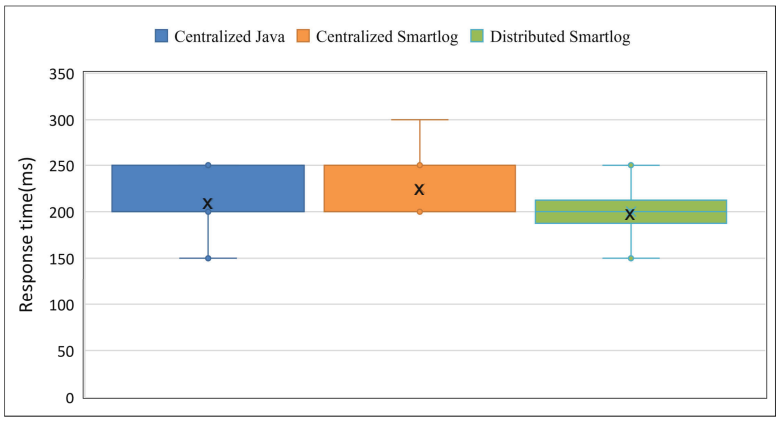


Figure 5.2: The average response time of each implementation.

To simplify for the following evaluations, the average value of the response time is used to represent the performance. As seen, the average response time of the distributed implementation is better than the average response time of both centralized implementations. It means that $\max(\sum_{i\in R_2} t_i, \sum_{j\in R_3} t_j) + Nt_c < \sum_{j\in N}\sum_{i\in R_j} t_i$, because, in the real-time simulation platform, the time delay t_c is not greater than the computing time (t_i) in the Raspberry Pis. Besides, the data sharing over the network makes the local query time faster than through a server. Moreover, the computing units work in parallel and share the computing load, which apparently reduces the response time (i.e. increases the execution performance).

5.3 Sensitivity to the number of computing units

As aforementioned, we proposed the CPDE methodology, and subsequently the SARD tool, to handle the increasing complexity of distributed programming when the number

of computing node scales up. With the CPDE, the same *Smartlog* program is used in the SARD whatever the number of nodes is. But the increasing number of participating computing nodes may have an impact on the response time of the distributed execution.

5.3.1 Methodology

We increase the number of PV systems and associated computing units in the network from 3 to 10, in order to observe its influence on the performance of the distributed execution. Our experimental infrastructure limits the number of Raspberry Pis to 10 (the maximum number of Raspberry Pis in our real-time simulation platform). We continue to perform the same test-case in the same experimental environment with the three implementations: centralized Java, centralized *Smartlog*, and distributed *Smartlog* obtained with the SARD. The average response time, which is determined as in Section 5.2, is once again used for the execution performance evaluation.

5.3.2 Results and discussion

The results of the experiment are shown in Figure 5.3.

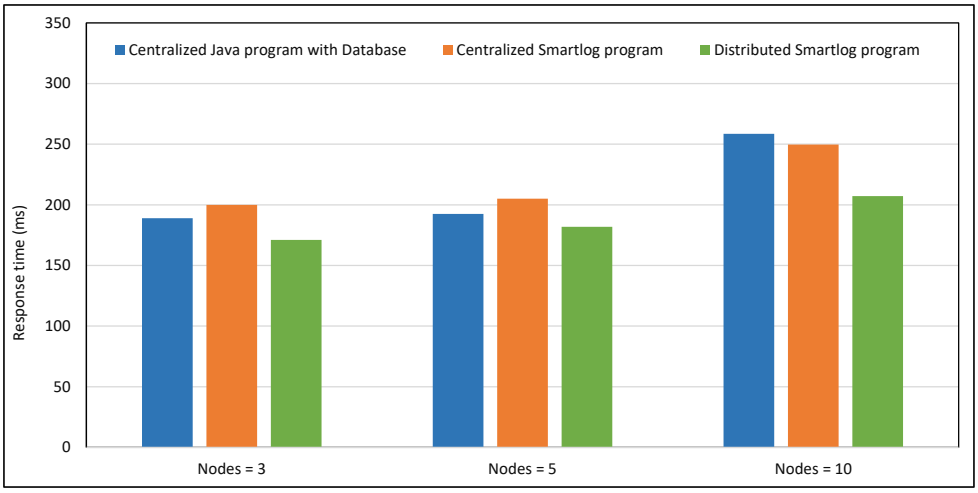


Figure 5.3: Impact of the number of nodes on the response time of the voltage regulation implementations.

In all situations (3, 5 and 10 computing nodes), the response time of the distributed *Smartlog* execution is better than both centralized ones. The gain in response time increases with the number of computing nodes. When the number of nodes is small, the response time of the distributed execution is just a bit smaller than the centralized

ones. Nevertheless, as the number of nodes increases, the difference in the response time of the three implementations is more and more significant. The response time of the distributed *Smartlog* execution is much better than the one of the centralized programs when the number of nodes equal to 10. Contrariwise, the response time of the centralized *Smartlog* is lower than the centralized Java one for 3 and 5 nodes. The situation is inverted for 10 nodes. Experimentation with more than 10 nodes is necessary to confirm or infirm this tendency.

The better performance of the distributed *Smartlog* execution can be justified using the following arguments. Calling $\bar{t_i}$ the average time to execute a rule, and $\bar{R_j}$ the average number of rules processed by a node. When N is big enough, so that $Nt_c \gg \max(\sum_{i \in R_2} t_i, \sum_{j \in R_3} t_j)$, the limitation of the response time of the distributed programming is:

$$\lim_{N}\{\max(\sum_{i \in R_2} t_i, \sum_{j \in R_3} t_j) + Nt_c\} = Nt_c \tag{5.2}$$

Meanwhile, the limit of the response time of a centralized programming is:

$$\lim_{N}\{\sum_{j \in N} \sum_{i \in R_j} t_i\} = N\bar{R_j}\bar{t_i} \tag{5.3}$$

With the same architecture of experimentation, so that $t_c < \bar{R_j}\bar{t_i}$, when the number of nodes in the network increases ($N \to \infty$), the response time of the centralized programming becomes much larger than the one of the distributed programming. That demonstrates the advantage of distributed programming regarding the scalability of the system. Besides, the distributed programs are automatically generated by the SARD tool. The increasing number of computing nodes does not affect the simplicity of the implementation.

5.4 Sensitivity to the distributed data configuration

The SARD tool operates at the level of the predefined data distribution. In this section, the sensitivity to the data distribution is considered. This work aims at confirming the correctness of the CPDE method with various data distributions as well as observing the effects of various distributed data configurations on the performance of the execution.

5.4.1 Methodology

To show the influence of the distributed data configurations, we use once again the last application, presented in Section 4.6. We focus mainly on the impact of the fragmentation and allocation of the intermediate `data_types` that are used to perform the aggregate computation over the network. If all the intermediate `data_types` are stored in the same place, this place is in charge of the computation of the aggregation functions. Otherwise, if we distribute these intermediate `data_types`, the aggregated computation is carried out in parallel by multiple nodes of the network. The different data distribution schemes of the intermediate `data_types` are expressed below:

Case 1: These intermediate `data_types` are stored in only one of the five Raspberry Pis (the second one in this case).

Case 2: Two intermediate `data_types` (`Warning` and `Alert`) are stored in the Raspberry Pi N^{o}7 and `Curtail` in the N^{o}9.

Case 3: We add a new Raspberry Pi on the 6^{th} node of the grid to store these intermediate `data_types`.

Case 4: Similar to **Case 2**, but `Alert` is fragmented vertically and stored in the N^{o}7 and N^{o}9 Raspberry Pis.

Only the first activation of the voltage regulation is considered to analyze the influence of the data configurations on the performances. We use again (5.1) to assess the response time of each data distribution configuration. Note that cases 1, 2 and 4 have five computing units, and case 3 uses six units; this new computing unit is called Raspberry Pi N^{o}6.

5.4.2 Results and discussion

Every configuration mentioned above is tested with the application used in Section 5.2. The two main purposes of this test-case are to show the influence of multiple data distribution configurations on the performances as well as the adaptation of the developed tool to multiple data distribution designs.

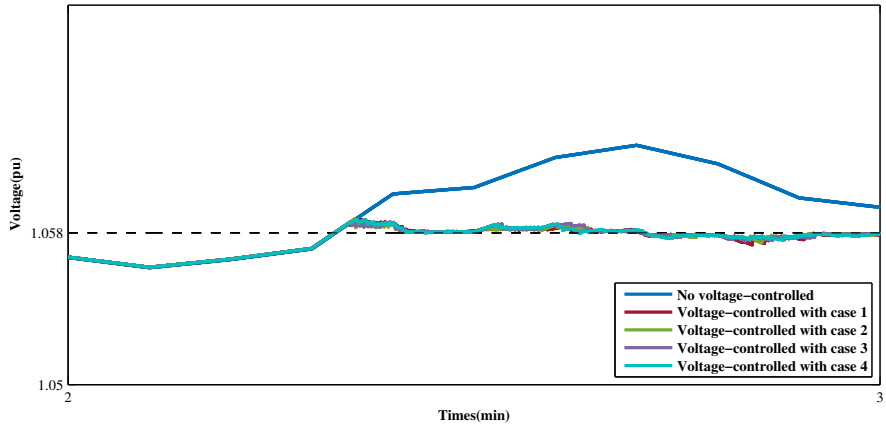


Figure 5.4: Responses at the 7^{th} PV node at the first moments of the voltage regulation with 4 schemes of data distribution.

The response of the four cases are presented in Figure 5.4. For all cases, the objective of the voltage regulation is reached. The voltage is controlled around the upper limit (1.058 pu) when there is PV over-production. Once again, we can conclude that our methodology can work well with many designs of data distribution, even with `data_types` that are fragmented horizontally or vertically. The centralized algorithm is always transformed exactly in the distributed programming.

As explained in Section 5.2, the response time in the experiment is collected based on the interval of consecutive changes of the power curtailment, once the power production increases linearly and causes over-voltage. We proceed with a statistic study of the four cases for the first moments, when the voltage regulation is activated during the experimentation. The results are placed next to the performance of the centralized implementations for comparison, as shown in Figure 5.5. The "X" mark presents the mean of the response time corresponding to each case study.

In Figure 5.5, the mean of response time of case 2 is the best one. That can be explained as follows. In case 1 and the centralized implementation, all the common data are stored at the same place. Intermediate computations based on common data are performed in only one node, which is why the response time is larger than in the other cases. In cases 2, 3, and 4, common data are allocated in multiple nodes. For case 3, another Raspberry Pi (N^{o}6) is in charge of the aggregated computation, but it takes more time than in case 2 because it requires more communications to request data from/to the various locations. The vertical data fragmentation in case 4 is also good for the average response time, which is smaller than cases 1, 3 and equal to case 2 in the distributed

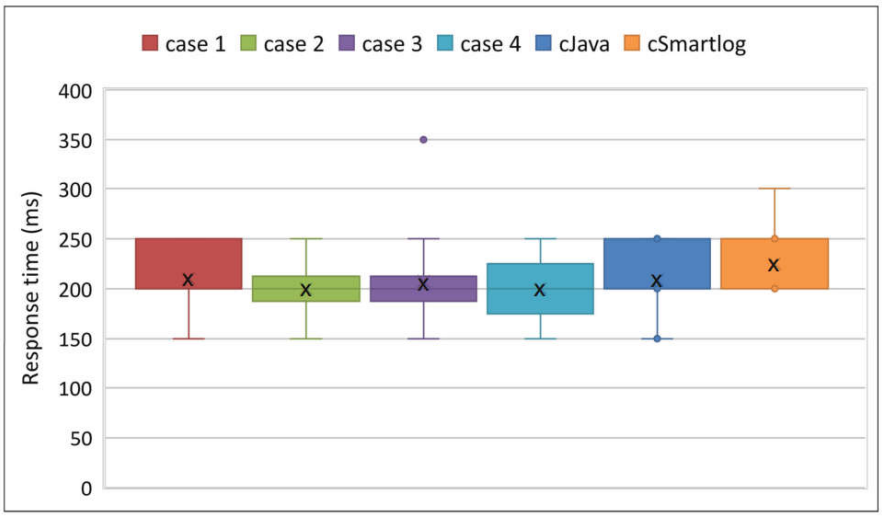


Figure 5.5: Responses time of all considered cases study.

implementation. However, the vertical fragmentation in case 4 seems to not fit in this application as case 4 has a big deviation compared to case 2. Table 5.2 presents the comparison of the four cases and the related response times expressions.

Table 5.2: Expression of the response time for each data distribution scheme.

Case	T_{comp}	T_{comm}
1	$\sum_{i \in R_{N^o7}} t_i$	$(2N-2)t_c$
2	$\max(\sum_{i \in R_{N^o7}} t_i, \sum_{j \in R_{N^o9}} t_j)$	Nt_c
3	$\max(\sum_{i \in R_j (j <> N^o6)} t_i, \sum_{i \in R_{N^o6}} t_i)$	$2Nt_c$
4	$\max(\sum_{i \in R_{N^o7}} t_i, \sum_{j \in R_{N^o9}} t_j)$	Nt_c

According to Table 5.2, the communication time of cases 2 and 4 are equal, but the number of fired rules in a node is not the same in each case. Case 4 has possibly longer computing time than case 2 because case 4 must deal with data integrity by regrouping vertical fragments. For case 3, the aggregate function is dispersed on another node bus, which needs more communications than in case 1. In fact, the comparison of their performance depends on the deviation of the average time delay of communications and the required time to execute a rule. This analysis reinforce the results shown in Figure 5.5 and presents the impact of the data distribution configuration on the response time of each distributed implementation.

As a result, the configuration of the data distribution has a strong influence on the performances of the execution. Thus, the problem of the optimal data distribution must

be addressed in the future to reach the best performances with the upcoming *Smartlog* implementations.

5.5 Conclusion

In this chapter, we analyzed various aspects of the distributed execution performances through an actual smart grid application. The performances of the distributed execution, the sensitivity to the number of computing units as well as to the data distribution configuration were analyzed and discussed in detail through the obtained results in the real-time simulation platform.

The CPDE provides a new approach of distributed deployment of smart grids applications. It deals with the scalability issue of the system and the sharing of computation load in local computing units. With the same experimental environment, the distributed programming promises to accelerate the response time compared to the centralized programming, even with imperative language.

The analysis also shows that the performances of the distributed execution depends on the data distribution. That leads to a question of choosing the best data distribution for each application to obtain the best performance. The answer to this question is the objective of perspective researches.

General conclusion and perspectives

The development of the smart grid scenario with full control and management even at the lowest level of the power grid poses the problem of managing a huge volume of sensor data as well as the scalability issue of the system. The traditional (centralized) paradigm is no longer sufficient to meet these requirements due to the high cost of server investments and the strengthening of the ICT infrastructure. In this context, the distributed paradigm seems to be more efficient because they deal with the imminent problems of control and management while ensuring the replacement of the conventional centralized controller. However, deploying control and management in the smart grid with a distributed paradigm remains a challenging task. In fact, distributed programming must perform centralized algorithms, while considering the various roles of the participants. It needs to manage the synchronization and message exchanges between participants in the network. These challenges increase with the growing number of participating computing devices.

The general context of power system development and the main challenges of smart grid scenarios are presented as the background of the thesis. The brief state of the art of distributed paradigm approaches such as distributed algorithms, multi-agent systems, distributed rule-based systems is taken into account. Based on that, we raised their contributions as well as their potential difficulties.

Regarding the needs for supervision and control in smart grids as well as lack of high-level programming abstraction in distributed programming, we abstracted the whole smart grid as a distributed database in which the declarative approach is suggested for distributed data manipulation in Chapter 2. Each participating computing device is in

charge of a network node that performs data manipulation and exchange data to other nodes. The architecture of each node in such an ad-hoc network is standardized as a rule-based system comprising four parts: network communication interface, sensors interface, local database, and rule engine. The prototype of implementation in this thesis used ProgreSQL as local databases and the triggers as a rule engine. *Smartlog* was proposed and developed as a new logic-based distributed data manipulation language dedicating to smart grids scenario. The general structure of the *Smartlog* language contains three parts: `data_type`s for declaring data schemes, `initial_data` for initializing values, and `Module`s for defining actions. *Smartlog*, based on Datalog, supports the declarative mechanism; reacting to changes in the environment (i.e. change in the sensor data) to adjust parameters (i.e. acting on actuators).

In Chapter 3, we aim at simplifying distributed programming in smart grids by totally hiding data exchange among computing devices. We propose to use Smartlog rule-based data manipulation language to program centralized control and management algorithms and then to automatically transform these centralized programs into sets of cooperating programs (i.e. to distribute the rules of the programs) in order to achieve distributed execution. This distributed programming methodology is called CPDE, for *Centralized Programming and Distributed Execution*. This methodology was implemented as SARD tool (Semi-automatic Rule Distribution). This name is used since the analysis of the method automatically distributes centralized *Smartlog* program into multiple ones but the data distribution description must be done manually. SARD expresses the data relationships in centralized rules by graphs. Based on the predefined data allocation, it decides on data routing in the graph by considering minimum communication costs. The output of this method is a set of distributed programs corresponding to data allocations. Test cases using the SARD tool are also proposed as incremental test to evaluate its behaviors.

The CPDE methodology was then applied to four typical applications in smart grids. The applications, from simple to complex, are local computations of load curve, incremental computation of bus admittance matrix, distributed secondary control in an isolated microgrid, and fair-sharing voltage control of distribution grid with high penetration of PV systems. All this served to show the adaptation of the intrinsic qualities of *Smartlog* in a smart grid context. The language presents a high-level programming abstraction, focusing on the expected result not on the processes, compactness, increment,

and simplicity. Besides, the execution time with *Smartlog* in the prototype was also assessed. The slower execution of data stored in the database gave a delayed response in control applications in comparison to data stored in read-only memory. However, that remained acceptable and sufficient for these near real-time control applications.

The performance of distributed *Smartlog* execution was also considered. The average response time in the last and most complex application was taken as a criterion for evaluating the performance of distributed and centralized execution (compared to a Java and a *Smartlog* centralized implementation). The result of the comparison is quite promising for the distributed *Smartlog* execution as it performs better than the others on a real-time simulation platform (OPAL-RT and Raspberry Pis). In addition, the sensitivity of the method to the increasing number of computing units and the change of data distribution design was discussed. The distributed *Smartlog* execution has the same advantage with the distributed paradigm: when the system scales up, the increase in performance is more visible. Also, the data distribution designs strongly affect the execution performance.

In this thesis, we have proposed a declarative approach for smart grids applications. Through experiments, we have highlighted evident qualities such as compactness, simplicity, reactivity, as well as a good performances of the distributed execution. The conclusions mentioned above only fulfills the first step of this prospective research. Many tasks still need to be developed and analyzed to complement the goal. The following perspectives are proposed for future researches:

Intermediate data placement: The data distribution description significantly affects the performance of the distributed deployment. It should provide a criterion to determine the best configuration for each application.

Optimizing the distribution of *Smartlog* program: in the CPDE method, we formulated the objective function of the communication decision with two criteria: the number of communications and the number of data-item transferred. However, the weight of the two criteria in the objective function is actually variable and depends mainly on the network characteristics. A cost model estimation for each specific network should be taken into account. Besides, other optimization techniques could be applied to deploy the distributed rules to improve the performances.

Data replication and duplication: In the distributed database design, data can be duplicated and replicated in the distributed system, but in Chapter 3, this problem is not automatically addressed. Thus, the method needs to be developed to support these cases, and to consider the synchronization costs of the copies to potentially increase the performances.

Rule engine improvement: In our development, PostgreSQL was chosen as a rule engine in which the *Smartlog* program can be executed in the computing devices. The main reason for choosing this DBMS are that it is open source and freely available. However, we know that the performance of this engine is not good in comparison with other DBMSs. A dedicated engine which doesn't have triggers, such as NoSQL-lite, is possible. Development in another DBMS may accelerate the execution performances if needed.

Bibliography

[1] Aarti Agarkar and Himanshu Agrawal. A review and vision on authentication and privacy preservation schemes in smart grid network. *Security and Privacy*, 2(2):e62, 2019.

[2] Ahmad Ahmad Kassem. *Programming networks with intensional destinations.* Theses, INSA de Lyon, November 2013.

[3] U. Ahsan and A. Bais. Distributed big data management in smart grid. In *2017 26th Wireless and Optical Communication Conference (WOCC)*, pages 1–6, April 2017.

[4] AIE. *World Energy Outlook 2018.* AIE, Paris, 2018.

[5] M. C. Alvarez-Herault, A. Labonne, S. Touré, T. Braconnier, V. Debusschere, R. Caire, and N. Hadjsaid. An original smart-grids test bed to teach feeder automation functions in a distribution grid. *IEEE Transactions on Power Systems*, 33(1):373–385, Jan 2018.

[6] Saeed Alyami, Yang Wang, Caisheng Wang, Junhui Zhao, and Bo Zhao. Adaptive real power capping method for fair overvoltage regulation of distribution networks with high penetration of pv systems. *IEEE Transactions on Smart Grid*, 5(6):2729–2738, 2014.

[7] Pierluigi Amodio and Francesca Mazzia. A parallel gauss–seidel method for block tridiagonal linear systems. *SIAM Journal on Scientific Computing*, 16(6):1451–1461, 1995.

[8] Mostafa M. Aref and Mohammed A. Tayyib. Lana-match algorithm: A parallel version of the rete-match algorithm. *Parallel Computing*, 24:763–775, 1998.

[9] Costin Bădică, Lars Braubach, and Adrian Paschke. Rule-based distributed and agent systems. In Nick Bassiliades, Guido Governatori, and Adrian Paschke, editors, *Rule-Based Reasoning, Programming, and Applications*, pages 3–28, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[10] David E Bakken, Anjan Bose, Carl H Hauser, David E Whitehead, and Gregary C Zweigle. Smart generation and transmission with coherent, real-time data. *Proceedings of the IEEE*, 99(6):928–951, 2011.

[11] Don Batory. The leaps algorithm. Technical report, University of Texas at Austin, Austin, TX, USA, 1994.

[12] Desong Bian, Murat Kuzlu, Manisa Pipattanasomporn, Saifur Rahman, and Di Shi. Performance evaluation of communication technologies and network structure for smart grid applications. *IET Communications*, 13(8):1025–1033, 2019.

[13] Ali Bidram, Ali Davoudi, Frank L Lewis, and Zhihua Qu. Secondary control of microgrids based on distributed cooperative control of multi-agent systems. *IET Generation, Transmission & Distribution*, 7(8):822–831, 2013.

[14] Biinform. Vietnam renewable energy report 2018. `https://biinform.com/Reports/1A6F-vietnam-renewable-energy-report-2018-7530.html`, march 2018.

[15] Stephen Boyd, Neal Parikh, Eric Chu, Borja Peleato, and Jonathan Eckstein. Distributed optimization and statistical learning via the alternating direction method of multipliers. *Found. Trends Mach. Learn.*, 3(1):1–122, January 2011.

[16] BP. Bp's statistical review of world energy 2019. `https://www.bp.com/content/dam/bp/en/corporate/pdf/energy-economics/statistical-review/bp-stats-review-2018-full-report.pdf`, july 2019.

[17] Lee Brownston, Robert Farrell, Elaine Kant, and Nancy Martin. *Programming Expert Systems in OPS5: An Introduction to Rule-based Programming.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1985.

[18] Jeffrey Considine, Feifei Li, George Kollios, and John Byers. Approximate aggregation techniques for sensor databases. In *Proceedings. 20th International Conference on Data Engineering*, pages 449–460. IEEE, 2004.

[19] José C Cunha, Pedro D Medeiros, Manuel B Carvalhosa, and Luís Moniz Pereira. Delta prolog: A distributed logic programming language and its implementation on distributed memory multiprocessors. In *Implementations of Distributed Prolog*. Citeseer, 1992.

[20] Houda Daki, Asmaa El Hannani, Abdelhak Aqqal, Abdelfattah Haidine, and Aziz Dahbi. Big data management in smart grid: concepts, requirements and implementation. *Journal of Big Data*, 4(1):13, Apr 2017.

[21] Koushan Das. Renewables in vietnam: Current opportunities and future outlook. `https://www.vietnam-briefing.com/news/vietnams-push-for-renewable-energy.html`, 2019. "[Online; accessed 17-March-2019]".

[22] JA Domínguez-Navarro, R Dufo-López, JM Yusta-Loyo, JS Artal-Sevil, and JL Bernal-Agustín. Design of an electric vehicle fast-charging station with integration of renewable energy and storage systems. *International Journal of Electrical Power & Energy Systems*, 105:46–58, 2019.

[23] Christian Dufour, Simon Abourida, and Jean Belanger. Hardware-in-the-loop simulation of power drives with rt-lab. In *Power Electronics and Drives Systems, 2005. PEDS 2005. International Conference on*, volume 2, pages 1646–1651. IEEE, 2005.

[24] Jonathan Eckstein and Wang Yao. Understanding the convergence of the alternating direction method of multipliers: Theoretical and computational perspectives. *Pacific Journal of Optimization*, 11:619–644, 10 2015.

[25] Jack Edmonds. Optimum branchings. *Journal of Research of the national Bureau of Standards B*, 71(4):233–240, 1967.

[26] Mustafa Amir Faisal, Zeyar Aung, John R. Williams, and Abel Sanchez. Securing advanced metering infrastructure using intrusion detection system with data stream mining. In Michael Chau, G. Alan Wang, Wei Thoo Yue, and Hsinchun Chen, editors, *Intelligence and Security Informatics*, pages 96–111, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[27] Klaus Fischer and Hans-Michael Windisch. Magsy: Ein regelbasiertes multiagentensystem. *KI*, 6:22–26, 1992.

[28] Harold N Gabow, Zvi Galil, Thomas Spencer, and Robert E Tarjan. Efficient algorithms for finding minimum spanning trees in undirected and directed graphs. *Combinatorica*, 6(2):109–122, 1986.

[29] Fei GAO, Ren KANG, Jun CAO, and Tao YANG. Primary and secondary control in dc microgrids: a review. *Journal of Modern Power Systems and Clean Energy*, 7(2):227–242, Mar 2019.

[30] Alan Gooding and David MacLeman. The application of active network management to the future smart grid. In *In proceeding Smart Grid 2010: Making it a reality, IET Conference*, pages 1–16, 01 2010.

[31] Gianluigi Greco, Antonella Guzzo, Domenico Sacc, and Francesco Scarcello. Event choice datalog: A logic programming language for reasoning in multiple dimensions. In *Proceedings of the 6th ACM SIGPLAN international conference on Principles and practice of declarative programming*, pages 238–249. ACM, 01 2004.

[32] Miniwatts Marketing Group. Internet usage statistics. `https://www.internetworldstats.com/stats.html`, june 2018. [Online; accessed 12-February-2019].

[33] NERC Sterring Group. technical analysis of august 14,2003 blackout. `https://www.nerc.com/docs/docs/blackout/NERC_Final_Blackout_Report_07_13_04.pdf`.

[34] Stéphane Grumbach and Fang Wang. Netlog, a rule-based language for distributed programming. In *International Symposium on Practical Aspects of Declarative Languages*, pages 88–103. Springer, 2010.

[35] Wei Gu, Guannan Lou, Wen Tan, and Xiaodong Yuan. A nonlinear state estimator-based decentralized secondary voltage control scheme for autonomous microgrids. *IEEE Transactions on Power Systems*, 2017.

[36] V. C. Gungor, D. Sahin, T. Kocak, S. Ergut, C. Buccella, C. Cecati, and G. P. Hancke. A survey on smart grid potential applications and communication requirements. *IEEE Transactions on Industrial Informatics*, 9(1):28–42, Feb 2013.

[37] Fanghong Guo, C Wen, and Y.-D Song. *Distributed control and optimization technologies in smart grid systems*. CRC Press, 01 2017.

[38] Nikos Hatziargyriou. *Microgrids: Architectures and Control*. John Wiley & Sons, 01 2014.

[39] Frederick Hayes-Roth. Rule-based systems. *Commun. ACM*, 28(9):921–932, September 1985.

[40] S. Y. Hui, C. K. Lee, and F. F. Wu. Electric springs a new smart grid technology. *IEEE Transactions on Smart Grid*, 3(3):1552–1561, Sep. 2012.

[41] P. Jesus, C. Baquero, and P. S. Almeida. A survey of distributed data aggregation algorithms. *IEEE Communications Surveys Tutorials*, 17(1):381–404, Firstquarter 2015.

[42] Renuka Kamdar, Priyanka Paliwal, and Yogendra Kumar. A state of art review on various aspects of multi-agent system. *Journal of Circuits, Systems and Computers*, 27:1830006, 01 2018.

[43] Simon Kemp. Digital in 2017: Southeast asia. `https://wearesocial.com/sg/blog/2017/02/digital-southeast-asia-2017`, 2017.

[44] Nguyen Quoc Khanh. Analysis of future generation capacity scenarios for vietnam. *Green Innovation and Development Centre (GreenID), Vietnam*, 2017.

[45] Javad Khazaei, Lingling Fan, Weiqing Jiang, and Durgesh Manjure. Distributed prony analysis for real-world pmu data. *Electric Power Systems Research*, 133:113 – 120, 2016.

[46] R. Klump, P. Agarwal, J. E. Tate, and H. Khurana. Lossless compression of synchronized phasor measurements. In *IEEE PES General Meeting*, pages 1–7, July 2010.

[47] Robert A. Kowalski. The early years of logic programming. *Commun. ACM*, 31(1):38–43, January 1988.

[48] Charles L. Forgy. Rete: A fast algorithm for the many pattern/many object pattern match problem. *Artificial Intelligence*, 19:17–37, 09 1982.

[49] La lettre des énergies renouvelables. Projet tiper en poitou-charentes - un mégawattheure solaire compétitif. `http://www.energies-renouvelables.org/articles/newsletter_15072014/poitou-tiper.asp`, 2014. [Accessed: March 2019].

[50] Leslie Lamport. Generalized consensus and paxos. 2005.

[51] Jeroen FJ Laros, André Blavier, Johan T den Dunnen, and Peter EM Taschner. A formalized description of the standard human variant nomenclature in extended backus-naur form. *BMC bioinformatics*, 12(4):S5, 2011.

[52] D.Hong LE. Building a compiler for smartlog language, june 2018.

[53] Jin Li. On peer-to-peer (p2p) content delivery. *Peer-to-Peer Networking and Applications*, 1(1):45–63, 2008.

[54] Xiao Li, Zhifang Wang, and Anna Scaglione. Decentralized data processing and management in smart grid via gossiping. In *2012 IEEE 7th Sensor Array and Multichannel Signal Processing Workshop (SAM)*, pages 1–4. IEEE, 2012.

[55] Chunfeng Liu, Yan Jiang, Feng Zhao, Qiao Sun, Yang Zhang, Zhiqi Li, and Chao Li. Design and applied research of the distributed real-time database in smart grid. In *16th International Conference on Advanced Communication Technology*, pages 719–724. IEEE, 2014.

[56] Boon Thau Loo, Tyson Condie, Minos Garofalakis, and etc. Declarative networking: language, execution and optimization. In *Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, pages 97–108. ACM, 2006.

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

[58] Xiaoqing Lu, Xinghuo Yu, Jingang Lai, Josep M Guerrero, and Hong Zhou. Distributed secondary voltage and frequency control for islanded microgrids with uncertain communication links. *IEEE Transactions on Industrial Informatics*, 13(2):448–460, 2017.

[59] Eng Keong Lua, Jon Crowcroft, Marcelo Pias, Ravi Sharma, Steven Lim, et al. A survey and comparison of peer-to-peer overlay network schemes. *IEEE Communications Surveys and tutorials*, 7(1-4):72–93, 2005.

[60] Eolas magazin. Smart grid evolution. `http://www.eolasmagazine.ie/smart-grid-evolution/`, 2018. Accessed: Tuesday, 22/01/2019 17:10 GMT+7.

[61] Daphne Mah, Peter Hills, Victor Li, and Richard Balme. *Smart Grid Applications and Developments*. Springer, 01 2014.

[62] A. Mehrizi-Sani. Chapter 2 - distributed control techniques in microgrids. In Magdi S. Mahmoud, editor, *Microgrid*, pages 43 – 62. Butterworth-Heinemann, 2017.

[63] Andreas Meier and Michael Kaufmann. *NoSQL Databases*, pages 201–218. Springer Fachmedien Wiesbaden, Wiesbaden, 2019.

[64] Daniel Miranker. *TREAT: A better match algorithm for AI production systems*. Artificial Intelligence Laboratory, University of Texas at Austin, 1987.

[65] Daniel K Molzahn, Florian Dörfler, Henrik Sandberg, Steven H Low, Sambuddha Chakrabarti, Ross Baldick, and Javad Lavaei. A survey of distributed optimization and control algorithms for electric power systems. *IEEE Transactions on Smart Grid*, 8(6):2941–2962, 2017.

[66] James Momoh. *Smart Grid: Fundamentals of Design and Analysis*. Wiley-IEEE Press, 2012.

[67] TTQ Nguyen, Ch Bobineau, Vincent Debusschere, Quang-Huy Giap, and N Hadj-Said. Using declarative programming for network data management in smart grids. In *Proceedings of the 22nd International Database Engineering & Applications Symposium*, pages 292–296. ACM, 2018.

[68] Diego Ongaro and John Ousterhout. In search of an understandable consensus algorithm. In *2014 {USENIX} Annual Technical Conference ({USENIX}{ATC} 14)*, pages 305–319, 2014.

[69] M Tamer Özsu and Patrick Valduriez. *Principles of distributed database systems*. Springer Science & Business Media, 2011.

[70] R. K. Pachauri and L. A. Meyer. Climate change 2014: Synthesis report. Technical report, IPCC, Geneva, Switzerland, 2014.

[71] Terence Parr. *The definitive ANTLR 4 reference*. Pragmatic Bookshelf, 2013.

[72] Arun G. PHADKE and Tianshu BI. Phasor measurement units, wams, and their applications in protection and control of power systems. *Journal of Modern Power Systems and Clean Energy*, 6(4):619–629, Jul 2018.

[73] C.Huy Pham. Wind and solar energy prospects in vietnam, 2018. [Online; accessed 13-December-2018].

[74] M. Pipattanasomporn, H. Feroze, and S. Rahman. Multi-agent systems in a distributed smart grid: Design and implementation. In *2009 IEEE/PES Power Systems Conference and Exposition*, pages 1–8, March 2009.

[75] Nagaraju Pogaku, Milan Prodanovic, and Timothy C Green. Modeling, analysis and testing of autonomous operation of an inverter-based microgrid. *IEEE Transactions on power electronics*, 22(2):613–625, 2007.

[76] S. Prasad and S. B. Avinash. Application of polyglot persistence to enhance performance of the energy data management systems. In *2014 International Conference on Advances in Electronics Computers and Communications*, pages 1–6, Oct 2014.

[77] Ellery E Queen. A discussion of smart meters and rf exposure issues. *Edison Electric Institute (EEI), Washington, DC, A Joint Project of the EEI and AEIC Meter Committees*, 2011.

[78] Nguyen Thanh Quynh. Implementation a distributed secondary control, May 2018.

[79] Rafiq Mahmud Rahat, Mohammad Hasan Imam, and Narottam Das. Comprehensive analysis of reliability and availability of sub-station automation system with iec 61850. In *2019 International Conference on Robotics, Electrical and Signal Processing Techniques (ICREST)*, pages 406–411. IEEE, 08 2019.

[80] Louiqa Raschid, Timos Sellis, and Alex Delis. A simulation-based study on the concurrent execution of rules in a database environment. *J. Parallel Distrib. Comput.*, 20:20–42, 01 1994.

[81] REN21. World energy outlook 2018. `http://www.ren21.net/wp-content/uploads/2018/06/17-8652_GSR2018_FullReport_web_final_.pdf`, 2018.

[82] B. A. Robbins, C. N. Hadjicostis, and A. D. Dom nguez-Garc a. A two-stage distributed architecture for voltage control in power distribution systems. *IEEE Transactions on Power Systems*, 28(2):1470–1482, May 2013.

[83] L. H. L. Rosa, N. Kagan, C. F. Meschini Almeida, and D. De Souza Pereira. A systemic approach for assessment of advanced distribution automation functionalities. *IEEE Transactions on Power Delivery*, pages 1–1, 2019.

[84] Andrzej Ruszczyński. On convergence of an augmented lagrangian decomposition method for sparse convex optimization. *Math. Oper. Res.*, 20(3):634–656, August 1995.

[85] H. Saadat. *Power System Analysis*. Number v. 1 in McGraw-Hill series in electrical and computer engineering. WCB/McGraw-Hill, 1998.

[86] Pierre Senellart, Louis Jachiet, Silviu Maniu, and Yann Ramusat. Provsql: Provenance and probability management in postgresql. *Proceedings of the VLDB Endowment*, 11(12):2034–2037, 2018.

[87] Q. Shafiee, J. C. Vasquez, and J. M. Guerrero. Distributed secondary control for islanded MicroGrids - A networked control systems approach. In *IECON 2012 - 38th Annual Conference on IEEE Industrial Electronics Society*, pages 5637–5642, October 2012.

[88] Andrew S. Tanenbaum and Maarten van Steen. *Distributed Systems: Principles and Paradigms (2Nd Edition)*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 2006.

[89] Tran Thi-Be. Assessing the potential of wind energy in coastal vietnam. `http://www.hus.vnu.edu.vn/files/LuanVan/LuanVan-TranThiBe-2013.pdf`, 2013.

[90] Chunming Tu, Xi He, Zhikang Shuai, and Fei Jiang. Big data issues in smart grid - a review. *Renewable and Sustainable Energy Reviews*, 79:1099 – 1107, 2017.

[91] Eben Upton and Gareth Halfacree. *Raspberry Pi user guide*. John Wiley & Sons, 2014.

[92] Chad Vicknair, Michael Macias, Zhendong Zhao, Xiaofei Nan, Yixin Chen, and Dawn Wilkins. A comparison of a graph database and a relational database: A data provenance perspective. In *Proceedings of the 48th Annual Southeast Regional Conference*, ACM SE '10, pages 42:1–42:6, New York, NY, USA, 2010. ACM.

[93] Vietnam Energy. The new regulation on tariffs of wind power projects in vietnam. `http://nangluongvietnam.vn/news/en/policy-planning/the-new-regulation-on-tariffs-of-wind-power-projects-in-vietnam.html`, 2019. [Accessed: March 2019].

[94] VietnamEnergy. Vietnam renewable energy development strategy, 2018. [Accessed: March 2019].

[95] T.N Trinh V.Q Nguyen. Electricity regulation in vietnam: Overview. `https://uk.practicallaw.thomsonreuters.com/4-628-5349?transitionType=Default&contextData=(sc.Default)`, 2019. "[Online; accessed 1-July-2019]".

[96] Jinghan Wang, Rui Zhou, Jing Li, and Guowei Wang. A distributed rule engine based on message-passing model to deal with big data. *Lecture Notes on Software Engineering*, 2:275–281, 01 2014.

[97] Yi Wang, Qixin Chen, Tao Hong, and Chongqing Kang. Review of smart meter data analytics: Applications, methodologies, and challenges. *IEEE Transactions on Smart Grid*, 2018.

[98] Yuan-Kang Wu, Shih Ming Chang, and Yi-Liang Hu. Literature review of power system blackouts. *Energy Procedia*, 141:428–431, may 2017.

[99] Lin Xiao, Stephen Boyd, and Sanjay Lall. A scheme for robust distributed sensor fusion based on average consensus. In *IPSN 2005. Fourth International Symposium on Information Processing in Sensor Networks, 2005.*, pages 63–70. IEEE, 2005.

[100] Yu Yan and Wencong Su. A fog computing solution for advanced metering infrastructure. In *2016 IEEE/PES Transmission and Distribution Conference and Exposition (T&D)*, pages 1–4, 05 2016.

[101] J. Zhang, J. Yang, and J. Li. When rule engine meets big data: Design and implementation of a distributed rule engine using spark. In *2017 IEEE Third International Conference on Big Data Computing Service and Applications (BigDataService)*, pages 41–49, April 2017.

[102] Dao Zhou, Jiahui Guo, Yingjie Zhang, Jidong Chai, Hesen Liu, Yong Liu, Can Huang, and Xun Gui. Distributed data analytics platform for wide-area synchrophasor measurement systems. *IEEE Transactions on Smart Grid*, 7:2397–2405, 2016.

Appendix A

Descriptions of the primary control in island microgrid

The structure of the VSI converter [75] in grid forming scenarios shown in Figure A.1 includes three main blocks: the power controller, the voltage controller and the current controller.

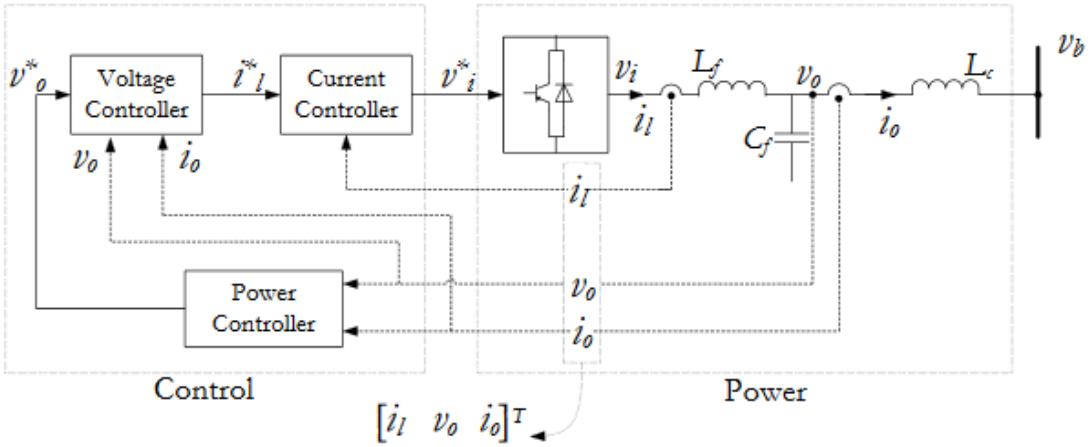


Figure A.1: The structure of control in VSI converter

The power controller uses the droop control method and has the structure proposed in Figure A.2.

The Voltage controller is computed by:

$$\begin{cases} i_{ld}^{*} = Fi_{od} - w_n C_f V_{oq} + K_{pv}(V_{od}^{*} - V_{od}) + K_{iv}\Phi_d \\ i_{lq}^{*} = Fi_{oq} - w_n C_f v_{od} + K_{pv}(v_{oq}^{*} - v_{oq}) + K_{iv}\Phi_q \end{cases} \tag{A.1}$$

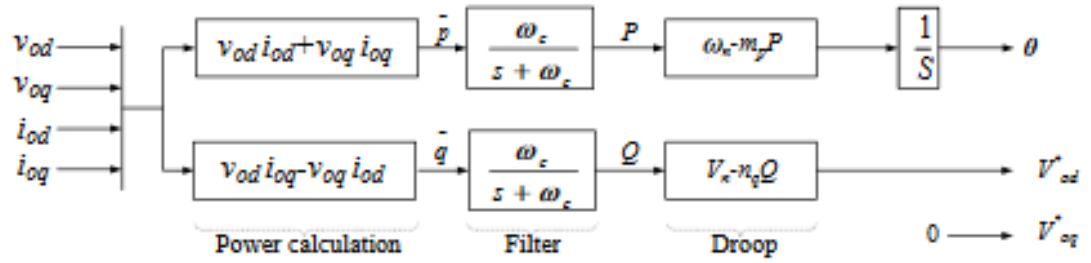


Figure A.2: The structure of power controller

With $\dot{\Phi}_d = v^*_{od} - v_{od}$; $\dot{\Phi}_q = v^*_{oq} - v_{oq}$.

The current controller is designed as bellow equations:

$$\begin{cases} v^*_{id} = v_{od} - w_n L_f i_{lq} + K_{pc}(i^*_{id} - i_{ld}) + K_{ic}\gamma_d \\ v^*_{iq} = v_{oq} - W_n L_f i_{ld} + K_{pc}(i^*_{iq} - i_{lq}) + K_{ic}\gamma_q \end{cases} \tag{A.2}$$

with $\dot{\gamma}_d = i^*_{od} - i_{od}$; $\dot{\gamma}_q = i^*_{oq} - i_{oq}$.

The parameters for the VSI converter are shown in Table A.1.

Table A.1: Parameters of VSI converter

Parameter	Symbol	Value
Maximum power	P	100kW
Voltage line to line	V_l	380V
Setpoint frequency	W_n	50 Hz
Maximum current	$I_{max} = 2P/3V$	175.44A
Basic impedance	$Z_b = V_l^2/P$	1.444 Ω
Continuous voltage	$V_{dc} = V_s * 2$	760V
Angular frequency	$W_g = 2 * \pi * W_n$	100π
Switch frequency	f_{sw}	10kHz
Desired attenuation	K_a	20%
Maximum variation of the network	x	5%
Basic capacity	$C_b = 1/(w_g * Z_b)$	2200μ F
Ripple current	$\Delta I_{max} = 10 I_{max}$	17.5A
Capacity	$C_f = x * C_b$	110μ F
Inductance 1	$L_f = V_{dc}/6 * f_{sw} * \Delta I_{max}$	7.22e-4H
Inductance 2	$L_c = \sqrt{1/K_a^2 + 1/(C_f f_{sw}^2)}$	1.36e-5H
Angular frequency of resonance	$w_{res} = \sqrt{(L_f + L_c)/(L_f L_c C_f}$	26096 rad/s
Frequency of resonance	$F_{res} = W_{res}/2\pi$	4155 Hz
Resistance	$R_f = 1/(2W_{res}C_f$	0.729 Ω

Appendix B

Executable test-cases and report

B.1 Centralized test

B.1.1 Rules containing only atoms and no fragmentation

```
Program(Test) {
Data_types{
A(AttrA1 : int key, AttrA2 : float, AttrA3
    : float).
B(AttrB1: int key, AttrB2 : float).
H(AttrH1 : int key, AttrH2 : float, AttrH3
    : float).}
Module(A){
H(i, m, n) :- A(i, m, k), B(i, n).}}
```

Listing B.1: Centralized SM program

```
IPmap {
"`S1"' : '192.168.1.101:30000';}
A(AttrA1 : int key, AttrA2 : float, AttrA3 : float) {
Fragment(A) {
Location : 'S1';}}.
B(AttrB1 : int key, AttrB2 : float, AttrB3 : float) {
Fragment(B) {
Location : 'S1';}}.
H(AttrH1 : int key, AttrH2 : float, AttrH3 : float) {
Fragment(H) {
Location : 'S1';}}.
```

Listing B.2: Data distribution

Output

```
Program(Test) {
Data_types{
A (AttrA1:int key, AttrA2:float, AttrA3:float).
B (AttrB1:int key, AttrB2:float).
H (AttrH1:int key, AttrH2:float, AttrH3:float).
}
Module(A){
H(i, m, n) :- A(i, m, k), B(i, n);
```

```
}
}
```

Listing B.3: Centralized test - generated program by SARD tool

B.2 Fragmentation tests

B.2.1 Horizontal fragmentation

Input

```
Program(Test) {
Data_types{
A(AttrA1 : int key, AttrA2 : float, AttrA3
    : float).
B(AttrB1: int key, AttrB2 : float).
H(AttrH1 : int key, AttrH2 : float, AttrH3
    : float).}
Module(A){
H(i, m, n) :- A(i, m, k), B(i, n).}}
```

Listing B.4: Centralized SM program

```
IPmap {
'S1' : '192.168.1.101:30000';}
A(AttrA1 : int key, AttrA2 : float, AttrA3 : float) {
Fragment(A1) {
Conditions : AttrA1>5;
Location : 'S1';}
Fragment(A2) {
Conditions : AttrA1 <=5;
Location : 'S1';}}.
B(AttrB1 : int key, AttrB2 : float, AttrB3 : float) {
Fragment(B) {
Location : 'S1';}}.
H(AttrH1 : int key, AttrH2 : float, AttrH3 : float) {
Fragment(H) {
Location : 'S1';}}.
```

Listing B.5: Data distribution

Output

```
Program(Test) {
Data_types{
A1 (AttrA1:int key, AttrA2:float, AttrA3:float).
A2 (AttrA1:int key, AttrA2:float, AttrA3:float).
B (AttrB1:int key, AttrB2:float).
H (AttrH1:int key, AttrH2:float, AttrH3:float).
}
Module(A1){
H(i, m, n) :- A1(i, m, k), B(i, n);
}
Module(A2){
H(i, m, n) :- A2(i, m, k), B(i, n);
```

```
}
}
```

Listing B.6: Horizontal fragmentation - generated program by SARD tool

B.2.2 Vertical fragmentation

Input

```
Program(Test) {
Data_types{
A(AttrA1 : int key, AttrA2 : float, AttrA3
    : float).
B(AttrB1: int key, AttrB2 : float).
H(AttrH1 : int key, AttrH2 : float, AttrH3
    : float).}
Module(A){
H(i, m, n) :- A(i, m, k), B(i, n).}}
```

Listing B.7: Centralized SM program

```
IPmap {
'S1' : '192.168.1.101:30000';}
A(AttrA1 : int key, AttrA2 : float, AttrA3 : float) {
Fragment(A1) {
Attributes : AttrA1, AttrA2;
Location : 'S1';}
Fragment(A2) {
Attributes : AttrA1, AttrA3;
Location : 'S1';}}.
B(AttrB1 : int key, AttrB2 : float, AttrB3 : float) {
Fragment(B) {
Location : 'S1';}}.
H(AttrH1 : int key, AttrH2 : float, AttrH3 : float) {
Fragment(H) {
Location : 'S1';}}.
```

Listing B.8: Data distribution

Output

```
Program(Test) {
Data_types{
A1 (AttrA1:int key, AttrA2:float).
A2 (AttrA1:int key, AttrA3:float).
B (AttrB1:int key, AttrB2:float).
H (AttrH1:int key, AttrH2:float, AttrH3:float).
}
Module(A1){
H(i, m, n) :- A1(i, m), A2(i, k), B(i, n);
}
Module(A2){
H(i, m, n) :- A2(i, k), A1(i, m), B(i, n);
}
}
```

Listing B.9: Vertical fragmentation -generated program by SARD tool

B.2.3 Hybrid fragmentation

Input

```
Program(Test) {
Data_types{
A(AttrA1 : int key, AttrA2 : float, AttrA3
    : float).
B(AttrB1: int key, AttrB2 : float).
H(AttrH1 : int key, AttrH2 : float, AttrH3
    : float).}
Module(A){
H(i, m, n) :- A(i, m, k), B(i, n).}}
```

Listing B.10: Centralized SM program

```
IPmap {
'S1' : '192.168.1.101:30000';}
A(AttrA1 : int key, AttrA2 : float, AttrA3 : float) {
Fragment(A1) {
Attributes : AttrA1, AttrA2;
Location : 'S1';}
Fragment(A2) {
Conditions: AttrA1>5;
Attributes : AttrA1, AttrA3;
Location : 'S1';}
Fragment(A3) {
Conditions: AttrA1<=5;
Attributes : AttrA1, AttrA3;
Location : 'S1';}}.
B(AttrB1 : int key, AttrB2 : float, AttrB3 : float) {
Fragment(B) {
Location : 'S1';}}.
H(AttrH1 : int key, AttrH2 : float, AttrH3 : float) {
Fragment(H) {
Location : 'S1';}}.
```

Listing B.11: Data distribution

Output

```
Program(Test) {
Data_types{
A1 (AttrA1:int key, AttrA2:float).
A3 (AttrA1:int key, AttrA3:float).
A2 (AttrA1:int key, AttrA3:float).
B (AttrB1:int key, AttrB2:float).
H (AttrH1:int key, AttrH2:float, AttrH3:float).
}
Module(A2){
H(i, m, n) :- A2(i, k), A1(i, m), B(i, n);
}
Module(A1){
H(i, m, n) :- A1(i, m), A2(i, k), B(i, n);
H(i, m, n) :- A1(i, m), A3(i, k), B(i, n);
}
Module(A3){
H(i, m, n) :- A3(i, k), A1(i, m), B(i, n);
}
```

```
}
```

Listing B.12: Hybrid fragmentation - generated program by SARD tool

B.3 Distribution test

B.3.1 Distributed body part

Input

```
Program(Test) {
Data_types{
A(AttrA1 : int key, AttrA2 : float, AttrA3
    : float).
B(AttrB1: int key, AttrB2 : float).
H(AttrH1 : int key, AttrH2 : float, AttrH3
    : float).}
Module(A){
H(i, m, n) :- A(i, m, k), B(i, n).}}
```

Listing B.13: Centralized SM program

```
IPmap {
'S1' : '192.168.1.101:30000';
'S2' : '192.168.1.102:30000';}
A(AttrA1 : int key, AttrA2 : float, AttrA3 : float) {
Fragment(A) {
Location : 'S1';}}.
B(AttrB1 : int key, AttrB2 : float, AttrB3 : float) {
Fragment(B) {
Location : 'S2';}}.
H(AttrH1 : int key, AttrH2 : float, AttrH3 : float) {
Fragment(H) {
Location : 'S2';}}.
```

Listing B.14: Data distribution

Output

```
Program(Test) {
Data_types{
A (AttrA1:int key, AttrA2:float, AttrA3:float).
IPMap (Name:string key, Address:string).
TmpR1A1G2 (AttrA2:float, AttrA1:int key).
}
Initial_data{
IPMap('S2','192.168.1.102:30000').
}
Module(A){
^TmpR1A1G2(m, i) :- A(i, m, k), IPMap('S2', @ip);
}
}
```

Listing B.15: Distributed body at site S1 - generated program by SARD tool

```
Program(Test) {
Data_types{
B (AttrB1:int key, AttrB2:float).
H (AttrH1:int key, AttrH2:float, AttrH3:float).
TmpR1A1G2 (AttrA2:float, AttrA1:int key).
}
Module(TmpR1A1G2){
H(i, m, n) :- !TmpR1A1G2(m, i), B(i, n);
}
}
```

Listing B.16: Distributed body at site S2 - generated program by SARD tool

B.3.2 Seperated head part

Input

```
Program(Test) {
Data_types{
A(AttrA1 : int key, AttrA2 : float, AttrA3
    : float).
B(AttrB1: int key, AttrB2 : float).
H(AttrH1 : int key, AttrH2 : float, AttrH3
    : float).}
Module(A){
H(i, m, n) :- A(i, m, k), B(i, n).}}
```

Listing B.17: Centralized SM program

```
IPmap {
'S1' : '192.168.1.101:30000';
'S2' : '192.168.1.102:30000';}
A(AttrA1 : int key, AttrA2 : float, AttrA3 : float) {
Fragment(A) {
Location : 'S1';}}.
B(AttrB1 : int key, AttrB2 : float, AttrB3 : float) {
Fragment(B) {
Location : 'S1';}}.
H(AttrH1 : int key, AttrH2 : float, AttrH3 : float) {
Fragment(H) {
Location : 'S2';}}.
```

Listing B.18: Data distribution

Output

```
Program(Test) {
Data_types{
A (AttrA1:int key, AttrA2:float, AttrA3:float).
IPMap (Name:string key, Address:string).
B (AttrB1:int key, AttrB2:float).
H (AttrH1:int key, AttrH2:float, AttrH3:float).
}
Initial_data{
```

```
IPMap('S2','192.168.1.102:30000').
}
Module(A){
^H(i, m, n) :- A(i, m, k), B(i, n), IPMap('S2', @ip);
}
}
```

Listing B.19: Seperated head part at site S1 - generated program by SARD tool

```
Program(Test) {
Data_types{
H (AttrH1:int key, AttrH2:float, AttrH3:float).
}
}
```

Listing B.20: Seperated head part at site S2 - generated program by SARD tool

B.4 Distribution fragmentation test

Input

```
Program(Test) {
Data_types{
A(AttrA1 : int key, AttrA2 : float, AttrA3
    : float).
B(AttrB1: int key, AttrB2 : float).
H(AttrH1 : int key, AttrH2 : float, AttrH3
    : float).}
Module(A){
H(i, m, n) :- A(i, m, k), B(i, n).}}
```

Listing B.21: Centralized SM program

```
IPmap {
'S1' : '192.168.1.101:30000';
'S2' : '192.168.1.102:30000';}
A(AttrA1 : int key, AttrA2 : float, AttrA3 : float) {
Fragment(A1) {
Attributes : AttrA1, AttrA2;
Location : 'S1';}
Fragment(A2) {
Conditions: AttrA1>5;
Attributes : AttrA1, AttrA3;
Location : 'S1';}
Fragment(A3) {
Conditions: AttrA1<=5;
Attributes : AttrA1, AttrA3;
Location : 'S2';}}.
B(AttrB1 : int key, AttrB2 : float, AttrB3 : float) {
Fragment(B) {
Location : 'S2';}}.
H(AttrH1 : int key, AttrH2 : float, AttrH3 : float) {
Fragment(H) {
Location : 'S1';}}.
```

Listing B.22: Data distribution

Output

```
Program(Test) {
Data_types{
A1 (AttrA1:int key, AttrA2:float).
A2 (AttrA1:int key, AttrA3:float).
IPMap (Name:string key, Address:string).
TmpR1A2G3 (AttrA1:int key, AttrA2:float).
TmpR1A3G2 (AttrA2:float, AttrA1:int key).
H (AttrH1:int key, AttrH2:float, AttrH3:float).
TmpR1A1G3 (AttrA1:int key, AttrA2:float).
}
Initial_data{
IPMap('S2','192.168.1.102:30000').
}
Module(A2){
^TmpR1A2G3(i, m) :- A2(i, k), A1(i, m), IPMap('S2', @ip);
}
Module(A1){
^TmpR1A1G3(i, m) :- A1(i, m), A2(i, k), IPMap('S2', @ip);
^TmpR1A3G2(m, i) :- A1(i, m), i<=5, IPMap('S2', @ip);
}
}
```

Listing B.23: At site S1 - generated program by SARD tool

```
Program(Test) {
Data_types{
A3 (AttrA1:int key, AttrA3:float).
IPMap (Name:string key, Address:string).
TmpR1A2G3 (AttrA1:int key, AttrA2:float).
TmpR1A3G2 (AttrA2:float, AttrA1:int key).
B (AttrB1:int key, AttrB2:float).
H (AttrH1:int key, AttrH2:float, AttrH3:float).
TmpR1A1G3 (AttrA1:int key, AttrA2:float).
}
Initial_data{
IPMap('S1','192.168.1.101:30000').
}
Module(TmpR1A1G3){
^H(i, m, n) :- !TmpR1A1G3(i, m), B(i, n), IPMap('S1', @ip);
}
Module(TmpR1A2G3){
```

```
^H(i, m, n) :- !TmpR1A2G3(i, m), B(i, n), IPMap('S1', @ip);
}
Module(TmpR1A3G2){
^H(i, m, n) :- !TmpR1A3G2(m, i), A3(i, k), B(i, n), IPMap('S1', @ip);
}
}
```

Listing B.24: At site S2 - generated program by SARD tool

B.5 Assignment tests without aggregate function

Input

```
Program(Test) {
Data_types{
A(AttrA1 : int key, AttrA2 : float, AttrA3
    : float).
B(AttrB1: int key, AttrB2 : int,
    AttrB3:float).
H(AttrH1 : int key, AttrH2 : float, AttrH3
    : float).}
Module(A){
H(i, m, l) :- A(i, m, k), B(i, j, n), l:=
    k+n.}}
```

Listing B.25: Centralized SM program

```
IPmap {
'S1' : '192.168.1.101:30000';
'S2' : '192.168.1.102:30000';}
A(AttrA1 : int key, AttrA2 : float, AttrA3 : float) {
Fragment(A1) {
Attributes : AttrA1, AttrA2;
Location : 'S1';}
Fragment(A2) {
Attributes : AttrA1, AttrA3;
Location : 'S2';}}.
B(AttrB1 : int key, AttrB2 : int, AttrB3 : float) {
Fragment(B1) {
Conditions: AttrB1 <5;
Location : 'S1';}
Fragment(B2) {
Conditions: AttrB1>=5;
Location : 'S2';}}.
H(AttrH1 : int key, AttrH2 : float, AttrH3 : float) {
Fragment(H) {
Location : 'S1';}}.
```

Listing B.26: Data distribution

Output

```
Program(Test) {
Data_types{
TmpR1A1G2 (AttrA1:int key, AttrA3:float).
A1 (AttrA1:int key, AttrA2:float).
B1 (AttrB1:int key, AttrB2:int, AttrB3:float).
IPMap (Name:string key, Address:string).
TmpR1A3G2 (AttrA2:float, AttrA1:int key).
```

```
H (AttrH1:int key, AttrH2:float, AttrH3:float).
}
Initial_data{
IPMap('S2','192.168.1.102:30000').
}
Module(TmpR1A1G2){
H(i, m, l) :- !TmpR1A1G2(i, k), A1(i, m), B1(i, _, n), l:=k+n.
}
Module(A1){
^TmpR1A3G2(m, i) :- A1(i, m), IPMap('S2', @ip);
}
}
```

Listing B.27: At site S1 -generated program by SARD tool

```
Program(Test) {
Data_types{
TmpR1A1G2 (AttrA1:int key, AttrA3:float).
B2 (AttrB1:int key, AttrB2:int, AttrB3:float).
A2 (AttrA1:int key, AttrA3:float).
IPMap (Name:string key, Address:string).
TmpR1A3G2 (AttrA2:float, AttrA1:int key).
H (AttrH1:int key, AttrH2:float, AttrH3:float).
}
Initial_data{
IPMap('S1','192.168.1.101:30000').
}
Module(A2){
^TmpR1A1G2(i, k) :- A2(i, k), IPMap('S1', @ip);
}
Module(TmpR1A3G2){
^H(i, m, l) :- !TmpR1A3G2(m, i), A2(i, k), B2(i, _, n), l:=k+n, IPMap('S1', @ip);
}
}
```

Listing B.28: At site S2 - generated program by SARD tool

B.6 Aggregate function

B.6.1 Without Fragmentation

Input

```
Program(Test) {
Data_types{
A(AttrA1 : int key, AttrA2 : float, AttrA3
    : float).
B(AttrB1: int key, AttrB2 : int key,
    AttrB3:float).
H(AttrH1 : int key, AttrH2 : float, AttrH3
    : float).}
Module(A){
H(i, m, l) :- A(i, m, k), B(i, j, n), l:=
    sum(n).}}
```

Listing B.29: Centralized SM program

```
IPmap {
'S1' : '192.168.1.101:30000';
'S2' : '192.168.1.102:30000';}
A(AttrA1 : int key, AttrA2 : float, AttrA3 : float) {
Fragment(A1) {
Location : 'S1';}}.
B(AttrB1 : int key, AttrB2 : int key, AttrB3 : float) {
Fragment(B1) {
Location : 'S2';}}.
H(AttrH1 : int key, AttrH2 : float, AttrH3 : float) {
Fragment(H) {
Location : 'S1';}}.
```

Listing B.30: Data distribution

Output

```
Program(Test) {
Data_types{
A1 (AttrA1:int key, AttrA2:float, AttrA3:float).
IPMap (Name:string key, Address:string).
H (AttrH1:int key, AttrH2:float, AttrH3:float).
TmpR1A1G2 (AttrA2:float, AttrA1:int key).
}
Initial_data{
IPMap('S2','192.168.1.102:30000').
}
Module(A1){
^TmpR1A1G2(m, i) :- A1(i, m, k), IPMap('S2', @ip);
}
}
```

Listing B.31: At site S1 - generated program by SARD tool

```
Program(Test) {
Data_types{
```

```
IPMap (Name:string key, Address:string).
B1 (AttrB1:int key, AttrB2:int key, AttrB3:float).
H (AttrH1:int key, AttrH2:float, AttrH3:float).
TmpR1A1G2 (AttrA2:float, AttrA1:int key).
}
Initial_data{
IPMap('S1','192.168.1.101:30000').
}
Module(TmpR1A1G2){
^H(i, m, l) :- !TmpR1A1G2(m, i), B1(i, j, n), l:=sum(n), IPMap('S1', @ip);
}
}
```

Listing B.32: At site S2 - generated program by SARD tool

B.6.2 With fragmentation

Input

```
Program(Test) {
Data_types{
A(AttrA1 : int key, AttrA2 : float, AttrA3
    : float).
B(AttrB1: int key, AttrB2 : int key,
    AttrB3:float).
H(AttrH1 : int key, AttrH2 : float, AttrH3
    : float).}
Module(A){
H(i, m, l) :- A(i, m, k), B(i, j, n), l:=
    sum(n).}}
```

Listing B.33: Centralized SM program

```
IPmap {
'S1' : '192.168.1.101:30000';
'S2' : '192.168.1.102:30000';}
A(AttrA1 : int key, AttrA2 : float, AttrA3 : float) {
Fragment(A1) {
Location : 'S1';}}.
B(AttrB1 : int key, AttrB2 : int key, AttrB3 : float) {
Fragment(B1) {
Conditions: AttrB2<5;
Location : 'S1';}
Fragment(B2)  {
Conditions: AttrB2 >=5;
Location: 'S2';}}.
H(AttrH1 : int key, AttrH2 : float, AttrH3 : float) {
Fragment(H) {
Location : 'S1';}}.
```

Listing B.34: Data distribution

Output

```
Program(Test) {
Data_types{
A1 (AttrA1:int key, AttrA2:float, AttrA3:float).
TmpH2 (AttrA1:int key, AttrA2:float, L:float).
IPMap (Name:string key, Address:string).
```

```
TmpH1 (AttrA1:int key, AttrA2:float, L:float).
B1 (AttrB1:int key, AttrB2:int key, AttrB3:float).
H (AttrH1:int key, AttrH2:float, AttrH3:float).
TmpR1A2G2 (AttrA1:int key, AttrA2:float).
}
Initial_data{
IPMap('S2','192.168.1.102:30000').
}
Module(A1){
TmpH1(i, m,l) :- A1(i, m, k), B1(i, j, n), l:=sum(n).
^TmpR1A2G2(i, m) :- A1(i, m, k), IPMap('S2', @ip);
}
Module(TmpH2){
H(i, m, l) :- !TmpH2(i, m,l1), !TmpH1(i, m,l2), l:=l1+l2.
}
Module(TmpH1){
H(i, m, l) :- !TmpH1(i, m,l1), !TmpH2(i, m,l2), l:=l1+l2.
}
}
```

Listing B.35: At site S1 - generated program by SARD tool

```
Program(Test) {
Data_types{
TmpH2 (AttrA1:int key, AttrA2:float, L:float).
IPMap (Name:string key, Address:string).
B2 (AttrB1:int key, AttrB2:int key, AttrB3:float).
TmpR1A2G2 (AttrA1:int key, AttrA2:float).
}
Initial_data{
IPMap('S1','192.168.1.101:30000').
}
Module(TmpR1A2G2){
^TmpH2(i, m,l) :- !TmpR1A2G2(i, m), B2(i, j, n), l:=sum(n), IPMap('S1', @ip);
}
}
```

Listing B.36: At site S2 - generated program by SARD tool

B.7 Condition test

B.7.1 With assignment

Input

```
Program(Test) {
Data_types{
A(AttrA1 : int key, AttrA2 : float, AttrA3
    : float).
B(AttrB1: int key, AttrB2 : int,
    AttrB3:float).
H(AttrH1 : int key, AttrH2 : float, AttrH3
    : float).}
Module(A){
H(i, m, l) :- A(i, m, k), B(i, j, n), l:=
    k+n, k>10.}}
```

Listing B.37: Centralized SM program

```
IPmap {
'S1' : '192.168.1.101:30000';
'S2' : '192.168.1.102:30000';}
A(AttrA1 : int key, AttrA2 : float, AttrA3 : float) {
Fragment(A1) {
Location : 'S1';}}.
B(AttrB1 : int key, AttrB2 : int key, AttrB3 : float) {
Fragment(B1) {
Conditions: AttrB2<5;
Location : 'S1';}
Fragment(B2)  {
Conditions: AttrB2 >=5;
Location: 'S2';}}.
H(AttrH1 : int key, AttrH2 : float, AttrH3 : float) {
Fragment(H) {
Location : 'S1';}}.
```

Listing B.38: Data distribution

Output

```
Program(Test) {
Data_types{
TmpR1A2G2 (AttrA2:float, AttrA1:int key, AttrA3:float).
A1 (AttrA1:int key, AttrA2:float, AttrA3:float).
IPMap (Name:string key, Address:string).
B1 (AttrB1:int key, AttrB2:int key, AttrB3:float).
H (AttrH1:int key, AttrH2:float, AttrH3:float).
}
Initial_data{
IPMap('S2','192.168.1.102:30000').
}
Module(A1){
H(i, m, l) :- A1(i, m, k), B1(i, j, n), l:=k+n, k>10.
^TmpR1A2G2(m, i, k) :- A1(i, m, k), k>10, IPMap('S2', @ip);
}
}
```

Listing B.39: At site S1 - generated program by SARD tool

```
Program(Test) {
Data_types{
TmpR1A2G2 (AttrA2:float, AttrA1:int key, AttrA3:float).
IPMap (Name:string key, Address:string).
H (AttrH1:int key, AttrH2:float, AttrH3:float).
B2 (AttrB1:int key, AttrB2:int key, AttrB3:float).
}
Initial_data{
IPMap('S1','192.168.1.101:30000').
}
Module(TmpR1A2G2){
^H(i, m, l) :- !TmpR1A2G2(m, i, k), B2(i, j, n), l:=k+n, k>10, IPMap('S1', @ip);
}
}
```

Listing B.40: At site S2 - generated program by SARD tool

B.7.2 with aggregate function

Input

```
Program(Test) {
Data_types{
A(AttrA1 : int key, AttrA2 : float, AttrA3
    : float).
B(AttrB1: int key, AttrB2 : int key,
    AttrB3:float).
H(AttrH1 : int key, AttrH2 : float, AttrH3
    : float).}
Module(A){
H(i, m, l) :- A(i, m, k), B(i, j, n),
    n>10, l:= sum(n).}}
```

Listing B.41: Centralized SM program

```
IPmap {
'S1' : '192.168.1.101:30000';
'S2' : '192.168.1.102:30000';}
A(AttrA1 : int key, AttrA2 : float, AttrA3 : float) {
Fragment(A1) {
Location : 'S1';}}.
B(AttrB1 : int key, AttrB2 : int key, AttrB3 : float) {
Fragment(B1) {
Conditions: AttrB2<5;
Location : 'S1';}
Fragment(B2)  {
Conditions: AttrB2 >=5;
Location: 'S2';}}.
H(AttrH1 : int key, AttrH2 : float, AttrH3 : float) {
Fragment(H) {
Location : 'S1';}}.
```

Listing B.42: Data distribution

Output

```
Program(Test) {
Data_types{
```

```
A1 (AttrA1:int key, AttrA2:float, AttrA3:float).
TmpH2 (AttrA1:int key, AttrA2:float, L:float).
IPMap (Name:string key, Address:string).
TmpH1 (AttrA1:int key, AttrA2:float, L:float).
B1 (AttrB1:int key, AttrB2:int key, AttrB3:float).
H (AttrH1:int key, AttrH2:float, AttrH3:float).
TmpR1A2G2 (AttrA1:int key, AttrA2:float).
}
Initial_data{
IPMap('S2','192.168.1.102:30000').
}
Module(A1){
TmpH1(i, m,l) :- A1(i, m, k), B1(i, j, n), n>10, l:=sum(n).
^TmpR1A2G2(i, m) :- A1(i, m, k), IPMap('S2', @ip);
}
Module(TmpH2){
H(i, m, l) :- !TmpH2(i, m,l1), !TmpH1(i, m,l2), l:=l1+l2.
}
Module(TmpH1){
H(i, m, l) :- !TmpH1(i, m,l1), !TmpH2(i, m,l2), l:=l1+l2.
}
}
```

Listing B.43: At site S1 - generated program by SARD tool

```
Program(Test) {
Data_types{
TmpH2 (AttrA1:int key, AttrA2:float, L:float).
IPMap (Name:string key, Address:string).
B2 (AttrB1:int key, AttrB2:int key, AttrB3:float).
TmpR1A2G2 (AttrA1:int key, AttrA2:float).
}
Initial_data{
IPMap('S1','192.168.1.101:30000').
}
Module(TmpR1A2G2){
^TmpH2(i, m,l) :- !TmpR1A2G2(i, m), B2(i, j, n), n>10, l:=sum(n), IPMap('S1', @ip);
}
}
```

Listing B.44: At site S2 - generated program by SARD tool

Appendix C

Distributed programs generated by the SARD tool

```
Program(VoltageRegulation) {
Data_types{
Actuator (ID:int key, Yield:float).
Slope (WarningID:int key, ID:int key, Slope:float, Vvar:float).
TmpR1Warning1G2 (WarningID:int key, Timestamp:int).
TmpR1Curtail1G2 (Pcur:float, ID:int key).
TmpCurtail1 (ID:int key, PRS:float, PMS:float).
Measure (ID:int key, Timestamp:int key, VOLT:float, POW:float, Pmax:float).
IPMap (Name:string key, Address:string).
Alert (ID:int key, Timestamp:int, Proportion:float).
WarningMeasure (WarningID:int key, ID:int key, VOLT:float, POW:float).
Plimit (WarningID:int key, ID:int key, PrefPower:float).
TmpR1Alert1G2 (Proportion:float, Timestamp:int, ID:int key).
Warning (WarningID:int key, Timestamp:int).
}
Initial_data{
IPMap('S2','192.168.1.102:30000').
IPMap('S3','192.168.1.103:30000').
}
Module(Measure){
^Warning(i, t) :- Measure(i, t, vi, p, pmi), vi>=1.042, ~WarningMeasure(i, i, _, _),
    IPMap('S2', @ip);
 :- Measure(i, t, vi, p, pmi), vi<1.042, !WarningMeasure(i, i, _, _);
```

```
^Alert(i, t, co) :- Measure(i, t, vi, p, pmi), vi>=1.047, co:=(1.058-vi)/(vi-1.042),
    WarningMeasure(i, i, _, _), IPMap('S2', @ip);
}
Module(TmpR1Warning1G2){
WarningMeasure(i, j, vj, pj) :- !TmpR1Warning1G2(i, t), Measure(j, t, vj, pj, _), j==1;
}
Module(TmpR1Alert1G2){
Slope(i, j, s, dv) :- TmpR1Alert1G2(co, t, i), Measure(j, t, vj, pj, pmj),
    WarningMeasure(i, j, vo, po), pj<>po, s:=(vj-vo)/(pj-po), s<>0, dv:=(co+1)*(vj-vo),
    j==1;
^TmpCurtail1(i,prs,pms) :- !TmpR1Alert1G2(co, t, i), Measure(j, t, _, _, pmj),
    Plimit(i, j, pli), pli:=least(pli,pmj), prs:=sum(pli), pms:=sum(pmj), IPMap('S3',
    @ip);
}
Module(Slope){
Plimit(i, j, pli) :- Slope(i, j, s, dv), WarningMeasure(i, j, _, po), pli:=po+dv/s,
    j==1;
}
Module(TmpR1Curtail1G2){
Actuator(j, pe) :- !TmpR1Curtail1G2(pe, i), Actuator(j, _), j==1;
}
}
```

Listing C.1: Distributed program for Raspberry N°5

```
Program(VoltageRegulation) {
Data_types{
Actuator (ID:int key, Yield:float).
Slope (WarningID:int key, ID:int key, Slope:float, Vvar:float).
TmpR1Warning1G2 (WarningID:int key, Timestamp:int).
TmpR1Curtail1G2 (Pcur:float, ID:int key).
TmpCurtail2 (ID:int key, PRS:float, PMS:float).
Measure (ID:int key, Timestamp:int key, VOLT:float, POW:float, Pmax:float).
Alert (ID:int key, Timestamp:int, Proportion:float).
IPMap (Name:string key, Address:string).
WarningMeasure (WarningID:int key, ID:int key, VOLT:float, POW:float).
Plimit (WarningID:int key, ID:int key, PrefPower:float).
TmpR1Alert1G2 (Proportion:float, Timestamp:int, ID:int key).
Warning (WarningID:int key, Timestamp:int).
}
Initial_data{
IPMap('S1','192.168.1.101:30000').
```

```
IPMap('S3','192.168.1.103:30000').
IPMap('S4','192.168.1.104:30000').
IPMap('S5','192.168.1.105:30000').
}
Module(Measure){
Warning(i, t) :- Measure(i, t, vi, p, pmi), vi>=1.042, ~WarningMeasure(i, i, _, _);
 :- Measure(i, t, vi, p, pmi), vi<1.042, !WarningMeasure(i, i, _, _);
Alert(i, t, co) :- Measure(i, t, vi, p, pmi), vi>=1.047, co:=(1.058-vi)/(vi-1.042),
    WarningMeasure(i, i, _, _);
}
Module(Warning){
^TmpR1Warning1G2(i, t) :- Warning(i, t), IPMap('S1', @ip);
WarningMeasure(i, j, vj, pj) :- Warning(i, t), Measure(j, t, vj, pj, _), j==2;
^TmpR1Warning1G2(i, t) :- Warning(i, t), IPMap('S3', @ip);
^TmpR1Warning1G2(i, t) :- Warning(i, t), IPMap('S4', @ip);
^TmpR1Warning1G2(i, t) :- Warning(i, t), IPMap('S5', @ip);
}
Module(Alert){
^TmpR1Alert1G2(co, t, i) :- Alert(i, t, co), IPMap('S1', @ip);
Slope(i, j, s, dv) :- Alert(i, t, co), Measure(j, t, vj, pj, pmj), WarningMeasure(i, j,
    vo, po), pj<>po, s:=(vj-vo)/(pj-po), s<>0, dv:=(co+1)*(vj-vo), j==2;
^TmpR1Alert1G2(co, t, i) :- Alert(i, t, co), IPMap('S3', @ip);
^TmpR1Alert1G2(co, t, i) :- Alert(i, t, co), IPMap('S4', @ip);
^TmpR1Alert1G2(co, t, i) :- Alert(i, t, co), IPMap('S5', @ip);
^TmpCurtail2(i,prs,pms) :- Alert(i, t, _), Measure(j, t, _, _, pmj), Plimit(i, j, pli),
    pli:=least(pli,pmj), prs:=sum(pli), pms:=sum(pmj), IPMap('S3', @ip);
}
Module(Slope){
Plimit(i, j, pli) :- Slope(i, j, s, dv), WarningMeasure(i, j, _, po), pli:=po+dv/s,
    j==2;
}
Module(TmpR1Curtail1G2){
Actuator(j, pe) :- !TmpR1Curtail1G2(pe, i), Actuator(j, _), j==2;
}
}
```

Listing C.2: Distributed program for Raspberry N°7

```
Program(VoltageRegulation) {
Data_types{
Actuator (ID:int key, Yield:float).
TmpCurtail5 (ID:int key, PRS:float, PMS:float).
```

```
TmpCurtail4 (ID:int key, PRS:float, PMS:float).
TmpR1Curtail1G2 (Pcur:float, ID:int key).
TmpCurtail3 (ID:int key, PRS:float, PMS:float).
TmpCurtail2 (ID:int key, PRS:float, PMS:float).
TmpCurtail1 (ID:int key, PRS:float, PMS:float).
Measure (ID:int key, Timestamp:int key, VOLT:float, POW:float, Pmax:float).
Alert (ID:int key, Timestamp:int, Proportion:float).
WarningMeasure (WarningID:int key, ID:int key, VOLT:float, POW:float).
Plimit (WarningID:int key, ID:int key, PrefPower:float).
TmpR1Alert1G2 (Proportion:float, Timestamp:int, ID:int key).
Curtail (ID:int key, Pcur:float).
Slope (WarningID:int key, ID:int key, Slope:float, Vvar:float).
TmpR1Warning1G2 (WarningID:int key, Timestamp:int).
IPMap (Name:string key, Address:string).
Warning (WarningID:int key, Timestamp:int).
}
Initial_data{
IPMap('S2','192.168.1.102:30000').
IPMap('S1','192.168.1.101:30000').
IPMap('S4','192.168.1.104:30000').
IPMap('S5','192.168.1.105:30000').
}
Module(Measure){
^Warning(i, t) :- Measure(i, t, vi, p, pmi), vi>=1.042, ~WarningMeasure(i, i, _, _),
    IPMap('S2', @ip);
 :- Measure(i, t, vi, p, pmi), vi<1.042, !WarningMeasure(i, i, _, _);
^Alert(i, t, co) :- Measure(i, t, vi, p, pmi), vi>=1.047, co:=(1.058-vi)/(vi-1.042),
    WarningMeasure(i, i, _, _), IPMap('S2', @ip);
}
Module(TmpR1Warning1G2){
WarningMeasure(i, j, vj, pj) :- !TmpR1Warning1G2(i, t), Measure(j, t, vj, pj, _), j==3;
}
Module(TmpR1Alert1G2){
Slope(i, j, s, dv) :- TmpR1Alert1G2(co, t, i), Measure(j, t, vj, pj, pmj),
    WarningMeasure(i, j, vo, po), pj<>po, s:=(vj-vo)/(pj-po), s<>0, dv:=(co+1)*(vj-vo),
    j==3;
TmpCurtail3(i,prs,pms) :- !TmpR1Alert1G2(co, t, i), Measure(j, t, _, _, pmj), Plimit(i,
    j, pli), pli:=least(pli,pmj), prs:=sum(pli), pms:=sum(pmj);
}
Module(TmpCurtail3){
```

```
Curtail(i, pe) :- !TmpCurtail3(i,prs1,pms6), !TmpCurtail2(i,prs2,pms7),
    !TmpCurtail5(i,prs3,pms8), !TmpCurtail4(i,prs4,pms9), !TmpCurtail1(i,prs5,pms10),
    prs:=prs1+prs2+prs3+prs4+prs5, pms:=pms6+pms7+pms8+pms9+pms10, pe:=prs/pms;
}
Module(TmpCurtail2){
Curtail(i, pe) :- !TmpCurtail2(i,prs1,pms6), !TmpCurtail3(i,prs2,pms7),
    !TmpCurtail5(i,prs3,pms8), !TmpCurtail4(i,prs4,pms9), !TmpCurtail1(i,prs5,pms10),
    prs:=prs1+prs2+prs3+prs4+prs5, pms:=pms6+pms7+pms8+pms9+pms10, pe:=prs/pms;
}
Module(TmpCurtail5){
Curtail(i, pe) :- !TmpCurtail5(i,prs1,pms6), !TmpCurtail3(i,prs2,pms7),
    !TmpCurtail2(i,prs3,pms8), !TmpCurtail4(i,prs4,pms9), !TmpCurtail1(i,prs5,pms10),
    prs:=prs1+prs2+prs3+prs4+prs5, pms:=pms6+pms7+pms8+pms9+pms10, pe:=prs/pms;
}
Module(TmpCurtail4){
Curtail(i, pe) :- !TmpCurtail4(i,prs1,pms6), !TmpCurtail3(i,prs2,pms7),
    !TmpCurtail2(i,prs3,pms8), !TmpCurtail5(i,prs4,pms9), !TmpCurtail1(i,prs5,pms10),
    prs:=prs1+prs2+prs3+prs4+prs5, pms:=pms6+pms7+pms8+pms9+pms10, pe:=prs/pms;
}
Module(TmpCurtail1){
Curtail(i, pe) :- !TmpCurtail1(i,prs1,pms6), !TmpCurtail3(i,prs2,pms7),
    !TmpCurtail2(i,prs3,pms8), !TmpCurtail5(i,prs4,pms9), !TmpCurtail4(i,prs5,pms10),
    prs:=prs1+prs2+prs3+prs4+prs5, pms:=pms6+pms7+pms8+pms9+pms10, pe:=prs/pms;
}
Module(Slope){
Plimit(i, j, pli) :- Slope(i, j, s, dv), WarningMeasure(i, j, _, po), pli:=po+dv/s,
    j==3;
}
Module(Curtail){
^TmpR1Curtail1G2(pe, i) :- Curtail(i, pe), IPMap('S1', @ip);
^TmpR1Curtail1G2(pe, i) :- Curtail(i, pe), IPMap('S2', @ip);
Actuator(j, pe) :- Curtail(i, pe), Actuator(j, _), j==3;
^TmpR1Curtail1G2(pe, i) :- Curtail(i, pe), IPMap('S4', @ip);
^TmpR1Curtail1G2(pe, i) :- Curtail(i, pe), IPMap('S5', @ip);
}
}
```

Listing C.3: Distributed program for Raspberry N°9

```
Program(VoltageRegulation) {
Data_types{
Actuator (ID:int key, Yield:float).
```

```
Slope (WarningID:int key, ID:int key, Slope:float, Vvar:float).
TmpR1Warning1G2 (WarningID:int key, Timestamp:int).
TmpCurtail4 (ID:int key, PRS:float, PMS:float).
TmpR1Curtail1G2 (Pcur:float, ID:int key).
Measure (ID:int key, Timestamp:int key, VOLT:float, POW:float, Pmax:float).
IPMap (Name:string key, Address:string).
Alert (ID:int key, Timestamp:int, Proportion:float).
WarningMeasure (WarningID:int key, ID:int key, VOLT:float, POW:float).
Plimit (WarningID:int key, ID:int key, PrefPower:float).
TmpR1Alert1G2 (Proportion:float, Timestamp:int, ID:int key).
Warning (WarningID:int key, Timestamp:int).
}
Initial_data{
IPMap('S2','192.168.1.102:30000').
IPMap('S3','192.168.1.103:30000').
}
Module(Measure){
^Warning(i, t) :- Measure(i, t, vi, p, pmi), vi>=1.042, ~WarningMeasure(i, i, _, _),
    IPMap('S2', @ip);
 :- Measure(i, t, vi, p, pmi), vi<1.042, !WarningMeasure(i, i, _, _);
^Alert(i, t, co) :- Measure(i, t, vi, p, pmi), vi>=1.047, co:=(1.058-vi)/(vi-1.042),
    WarningMeasure(i, i, _, _), IPMap('S2', @ip);
}
Module(TmpR1Warning1G2){
WarningMeasure(i, j, vj, pj) :- !TmpR1Warning1G2(i, t), Measure(j, t, vj, pj, _), j==4;
}
Module(TmpR1Alert1G2){
Slope(i, j, s, dv) :- TmpR1Alert1G2(co, t, i), Measure(j, t, vj, pj, pmj),
    WarningMeasure(i, j, vo, po), pj<>po, s:=(vj-vo)/(pj-po), s<>0, dv:=(co+1)*(vj-vo),
    j==4;
^TmpCurtail4(i,prs,pms) :- !TmpR1Alert1G2(co, t, i), Measure(j, t, _, _, pmj),
    Plimit(i, j, pli), pli:=least(pli,pmj), prs:=sum(pli), pms:=sum(pmj), IPMap('S3',
    @ip);
}
Module(Slope){
Plimit(i, j, pli) :- Slope(i, j, s, dv), WarningMeasure(i, j, _, po), pli:=po+dv/s,
    j==4;
}
Module(TmpR1Curtail1G2){
Actuator(j, pe) :- !TmpR1Curtail1G2(pe, i), Actuator(j, _), j==4;
}
```

```
}
```

Listing C.4: Distributed program for Raspberry N°12

```
Program(VoltageRegulation) {
Data_types{
Actuator (ID:int key, Yield:float).
TmpCurtail5 (ID:int key, PRS:float, PMS:float).
Slope (WarningID:int key, ID:int key, Slope:float, Vvar:float).
TmpR1Warning1G2 (WarningID:int key, Timestamp:int).
TmpR1Curtail1G2 (Pcur:float, ID:int key).
Measure (ID:int key, Timestamp:int key, VOLT:float, POW:float, Pmax:float).
IPMap (Name:string key, Address:string).
Alert (ID:int key, Timestamp:int, Proportion:float).
WarningMeasure (WarningID:int key, ID:int key, VOLT:float, POW:float).
Plimit (WarningID:int key, ID:int key, PrefPower:float).
TmpR1Alert1G2 (Proportion:float, Timestamp:int, ID:int key).
Warning (WarningID:int key, Timestamp:int).
}
Initial_data{
IPMap('S2','192.168.1.102:30000').
IPMap('S3','192.168.1.103:30000').
}
Module(Measure){
^Warning(i, t) :- Measure(i, t, vi, p, pmi), vi>=1.042, ~WarningMeasure(i, i, _, _),
    IPMap('S2', @ip);
 :- Measure(i, t, vi, p, pmi), vi<1.042, !WarningMeasure(i, i, _, _);
^Alert(i, t, co) :- Measure(i, t, vi, p, pmi), vi>=1.047, co:=(1.058-vi)/(vi-1.042),
    WarningMeasure(i, i, _, _), IPMap('S2', @ip);
}
Module(TmpR1Warning1G2){
WarningMeasure(i, j, vj, pj) :- !TmpR1Warning1G2(i, t), Measure(j, t, vj, pj, _), j==5;
}
Module(TmpR1Alert1G2){
Slope(i, j, s, dv) :- TmpR1Alert1G2(co, t, i), Measure(j, t, vj, pj, pmj),
    WarningMeasure(i, j, vo, po), pj<>po, s:=(vj-vo)/(pj-po), s<>0, dv:=(co+1)*(vj-vo),
    j==5;
^TmpCurtail5(i,prs,pms) :- !TmpR1Alert1G2(co, t, i), Measure(j, t, _, _, pmj),
    Plimit(i, j, pli), pli:=least(pli,pmj), prs:=sum(pli), pms:=sum(pmj), IPMap('S3',
    @ip);
}
Module(Slope){
```

```
Plimit(i, j, pli) :- Slope(i, j, s, dv), WarningMeasure(i, j, _, po), pli:=po+dv/s,
    j==5;
}
Module(TmpR1Curtail1G2){
Actuator(j, pe) :- !TmpR1Curtail1G2(pe, i), Actuator(j, _), j==5;
}
}
```

Listing C.5: Distributed program for Raspberry N°14

Une nouvelle approach pour la programmation distribuée dans les smartgrids

Thèse de Doctorant – Université Grenoble Alpes

Thi-Thanh-Quynh NGUYEN

Résumé – Cette thèse a pour objectif de développer le mécanisme de supervision et de contrôle distribué dans le réseau intelligent, dans lequel le système est composé d'une multitude d'unités de calcul coopérantes, et les calculs sont effectués à proximité des producteurs et des utilisateurs de données. Le défi de la programmation distribuée est de savoir comment gérer l'échange de données et la synchronisation des unités participantes lorsque le système évolue.

Dans cette thèse, nous proposons une approche innovante de la programmation d'un niveau d'abstraction élevé masquant ces difficultés. Premièrement, nous suggérons d'abriter toutes les unités de calcul Smartgrid (compteurs intelligents, capteurs, concentrateurs de données, etc.) sous forme de base de données distribuée. Chaque unité de calcul héberge une base de données locale et seules les données nécessaires à la poursuite du calcul sont échangées avec d'autres unités, ce qui diminue l'utilisation de la bande passante disponible. Par ailleurs, nous proposons également SmartLog, un langage basé sur des règles basé sur le langage Datalog) à dédier à la programmation d'applications de contrôle et de gestion en répondant immédiatement à toute modification des données. Nous proposons ensuite une méthodologie pour simplifier le programme distribué, appelée CPDE, comprenant deux étapes: (i) programer l'application centralisée en Smartlog et (ii) traduire le programme centralisé en programmes distribués en se basant sur la localisation réelle des données.

L'évaluation de la méthode basée sur les résultats de la simulation en temps réel montre sa adaptation à de nombreuses configurations de distribution de données, capables de programmer avec un niveau d'abstraction plus élevé, de résoudre le problème de scalabilité ainsi que de simplifier les implémentations en temps réel.

Mots-clés: smartgrids, micro-réseau, base de données distribuées, langage déclaratif et réactif, contrôle distribué, programmation distribuée.

A new approach of distributed programming in smartgrids

Ph.D Thesis – Grenoble Alpes University

Thi-Thanh-Quynh NGUYEN

Abstract – This thesis aims to develop the distributed supervision and control mechanism in smart grid in which the system is made up of a multitude of co-operating computing units, and calculations are performed close to the producers and consumer of data. The challenge of distributed programming is how to handle data exchange and the synchronization of the participating units when the system evolves.

In this thesis, we propose an innovative approach to programming of a high level of abstraction masking these difficulties. Firstly, we suggest to abstract all Smartgrid computing units (smart meters, sensors, data concentrators, etc.) as a distributed database. Each computing unit hosts a local database and only the data needed to continue the calculation are exchanged with other units, which decreases the use of the available bandwidth. Besides, we also propose SmartLog, a rule-based language based on the Datalog language) to dedicate to the programming of control and management applications by immediately responding to any changes in the data. We then propose a methodology to simplify the distributed program, named CPDE consisting of two steps: (i) programming the centralised application and (ii) translating the centralised program into distributed programs based on the actual location of data.

The evaluation of the method based on the real-time simulation results shows its adaptation with many data distribution configurations, capable of programming with a high level of abstraction, settlement of scalability issue as well as simplicity for implementations in real-time.

Keywords: Smart grid, microgrid, distributed supervision and control, distributed database, declarative and reactive language, distributed system, distributed programming.

Grenoble Electrical Engineering laboratory (G2Elab)
Bâtiment GreEn-ER, 21 avenue des Martyrs, F-38031 Grenoble Cedex 1, France
Laboratoire d'Informatique de Grenoble (LIG)
Bâtiment IMAG, 700 Avenue Centrale, 38401 Saint-Martin-d'Hères, France