A Service-Oriented, Cyber-Physical Reference Model for Smart Grid

Muhammad Umer Tariq, Santiago Grijalva, and Marilyn Wolf

Abstract. This chapter presents a cyber-physical reference model for smart grid. Most of the early smart grid applications have been developed in an ad-hoc manner, without any underlying framework. The proposed reference model addresses this issue and enables the design of smart grid as a robust system that is extensible to the future. The proposed reference model is based on service-oriented computing paradigm and is compatible with the existing service-oriented technologies, used in enterprise computing, such as Web Services. However, it also extends these technologies for handling the hard real-time aspects of smart grid by introducing resource-aware service deployment and quality-of-service (QoS)-aware service monitoring phases. According to the proposed reference model, each smart grid scenario is characterized by three elements: (1) an *application model* that describes the smart grid applications to be supported by the system as a set of resource- and QoS-aware service descriptions, (2) a *platform model* that describes the smart grid platform as a set of computing nodes, communication links, sensors, actuators, and power system entities, and (3) a set of algorithms that enable resource-aware service deployment, QoS-aware service discovery, and QoS-aware service monitoring. This chapter also presents typical development steps of a smart grid application according to the proposed reference model. Moreover, this chapter identifies a number of technological requirements that can enable the development of smart grid applications according to the proposed reference model. Although the development of these required technologies is a topic of ongoing research, this chapter identifies some potential solution approaches, based on state-of-the-art techniques from real-time systems literature. The case study of a demand response application has been employed to explain the various aspects of the proposed smart grid reference model.

Muhammad Umer Tariq · Santiago Grijalva · Marilyn Wolf
Georgia Institute of Technology, Atlanta
e-mail: {m.umer.tariq,sgrijalva,marilyn.wolf}@gatech.edu

1 Introduction

Traditional electric power grid is not capable of reliably handling the imminent deployment of large amounts of renewable energy sources because of their intermittent nature [10][18]. This deficiency has resulted in a worldwide effort towards realizing the vision of a smarter electric power grid [16]. This vision of a *smart grid* proposes to overlay the electric grid with a more extensive computation and communication infrastructure. Unfortunately, most of the early smart grid applications have been developed in an ad-hoc manner, without any underlying framework. This lack of an underlying framework has resulted in a set of isolated smart grid technologies, standards, and applications that are difficult to integrate and extend for the future [13].

In the past, various other complex engineering domains have faced similar problems in their early days. Research communities for those engineering domains overcame this problem by developing a *reference model* for the domain that could enable clear communication among different stakeholders and inform the development of an integrated set of technologies and standards for that domain [19] [14]. In this chapter, we leverage this idea of a reference model and propose a cyber-physical reference model for the domain of smart grid. The proposed reference model for smart grid will enable the development of smart grid technologies, standards, and applications in a robust, integrated, and flexible manner.

The proposed reference model for smart grid is based on service-oriented computing (SOC) paradigm, as this paradigm is uniquely capable of handling the large scale, open nature, and long lifecycle of smart grid scenarios. However, the traditional SOC paradigm, used in enterprise computing domain through popular technologies such as Web Services, cannot be directly applied to smart grid, because this traditional paradigm is not capable of handling the hard real-time aspects of smart grid [4]. Therefore, the proposed smart grid reference model extends the traditional SOC paradigm by introducing resource-aware service deployment and QoS-aware service monitoring phases. According to the proposed service-oriented reference model for smart gird, each smart grid scenario is characterized by three elements:

1. An *application model* that describes the smart grid applications to be supported by the system as a set of resource- and QoS-aware service descriptions.
2. A *platform model* that describes the smart grid platform as a set of computing nodes, communication links, sensors, actuators, and power system entities.
3. A set of algorithms that achieve resource-aware service deployment, QoS-aware service discovery, and QoS-aware service monitoring.

The proposed reference model is capable of describing relevant characteristics of a wide set of smart grid scenarios as it has a sufficiently rich set of features. Moreover, the proposed reference model is generic enough to be reconciled with the existing smart grid standards and technologies, but still provides valuable guidance for the evolution of these standards and technologies into an integrated and consistent set of future smart grid standards and technologies.

In this chapter, we explain various elements of the proposed reference model and present typical development steps of a smart grid application according to the proposed reference model. We also identify some technologies that must be developed before the proposed smart grid reference model could be applied in practice. Although the development of these technologies is a work-in-progress, we present promising solution approaches, based on some state-of-the-art techniques from real-time systems literature. In this chapter, we have used the case study of a demand response scenario in order to explain various aspects of the proposed reference model.

The rest of the chapter is organized as follows. Section 2 explains what is a reference model and how it can be employed successfully for the domain of smart grid. Section 3 presents the details of the proposed reference model for smart grid. This section also presents a development methodology for smart grid applications according to the proposed reference model. Section 4 identifies some technological requirements of the proposed reference model and presents some promising approaches for meeting those requirements. Section 5 presents the case study of a demand response scenario. Section 6 presents the conclusion.

2 What Is a Reference Model?

A reference model for a domain is an abstract conceptual framework, consisting of a small number of interlinked and unifying concepts for that domain. A reference model is designed to enable clear communication about the domain among various stakeholders. A reference model is not a standard or implementation technology in itself. However, it does ”inform” the development of a set of compatible standards and technologies for a certain domain [14] [2].

In the past, the concept of a reference model has been successfully employed in various domains to enable the development of a coherent set of technologies and standards for that domain. Following are some examples of reference models developed for various domains:

- Open Systems Interconnection (OSI) Reference Model for communication systems [19].
- Agent Systems Reference Model (ASRM) for multi-agent systems [15].
- National Institute of Standards and Technology (NIST) Reference Model for software engineering environments [2].
- National Institute of Standards and Technology (NIST) Reference Model for project support environments [1].
- Task-based Reference Model for real-time computer systems [12].

Similarly, the development of an appropriate reference model for smart grid can not only ensure clear communication among different stakeholders, but also help in the process of developing a coherent and consistent set of standards and technologies for smart grid. However, any reference model for smart grid must be based on concepts that are generic enough to be reconciled with existing standards (such as various NIST and IEC standards related to smart grid [13]) and technologies (such as real-time operating system and middleware [7] [11]), but still provide valuable guidance for the evolution of existing standards and technologies into a consistent and coherent set of future standards and technologies. In this chapter, we propose a reference model for smart grid that relies on the unification of concepts from the domains of service-oriented computing [4] and cyber-physical systems [17] [9] [8].

3 Reference Model for Smart Grid

This section presents the details of the proposed reference model for smart grid. The proposed reference model is based on service-oriented computing paradigm. Although service-oriented computing paradigm is currently being used in enterprise computing through Web Services technology, it cannot be directly applied to the domain of smart grid because of the hard real-time aspects of smart grid applications. On the other hand, the task-based reference model used in typical distributed, real-time systems, such as automotive and avionics, cannot be directly applied to the domain of smart grid as this reference model is not capable of handling the large scale, open nature, and long lifecycle of smart grid scenarios [12]. Our proposed reference model essentially extends the traditional service-oriented computing paradigm by introducing resource-aware service deployment and QoS-aware service monitoring phases.

According to the proposed reference model for smart gird, each smart grid scenario is characterized by three elements:

1. An *application model* that describes the smart grid applications to be supported by the system as a set of resource- and QoS-aware service descriptions.
2. A *platform model* that describes the smart grid platform as a set of computing nodes, communication links, sensors, actuators, and power system entities.
3. A *set of algorithms* that achieve resource-aware service deployment, QoS-aware service discovery, and QoS-aware service monitoring.

Figure 1 shows all the three elements of the proposed service-oriented reference model for smart grid. Figure 2 shows the major steps involved in the development

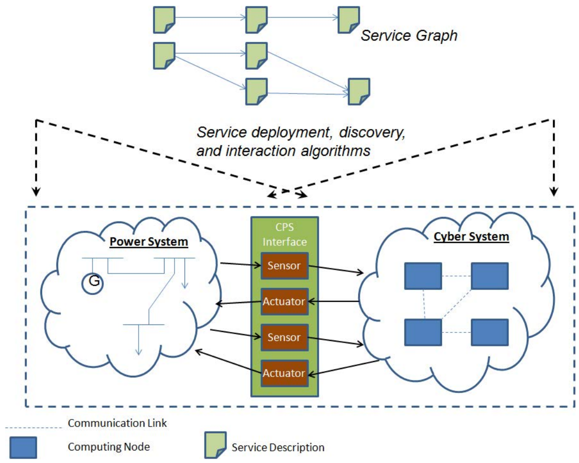

Fig. 1 Reference model for smart grid

of smart grid application according to the proposed reference model. In the *platform porting* step, a generic, service-based computing platform is ported to all the heterogeneous computing nodes involved in a smart grid scenario. In the *service modeling* step, the smart grid application is modeled as a set of services that interact with each other as well as with physical entities through sensing and actuation. In the *service implementation* phase, the implementation code for the services is developed. Both *service modeling* and *service implementation* steps contribute to the development of service descriptions. The service description of a service not only defines the messages that a service exchanges with other services, but it also defines sensing and control actions that the service takes on the co-located physical entities. Moreover, a service description identifies the quality-of-service (QoS) constraints on message exchanges with other services and platform resource requirements of a service. A service description also identifies various modes of operation of a service for various QoS fault scenarios.

In the *service deployment* phase, all the services are deployed on their associated computing nodes. This leads to the *service discovery* step, where all the services involved discover their peer services. This step could be performed online or offline depending on the nature of the smart grid application. In the *service interaction* step, services involved in a smart grid application interact by sending messages to each other. During the *service interaction* step, services switch between different modes of operation if QoS faults occur. Finally, through a *service update* phase, this smart grid reference model supports system maintenance and system updates. In the *service update* phase, services involved in the smart grid application are updated. These services again pass through *service implementation* and *service deployment* steps. Again, the *service update* step can be designed to work online or offline depending on the nature of smart grid application.

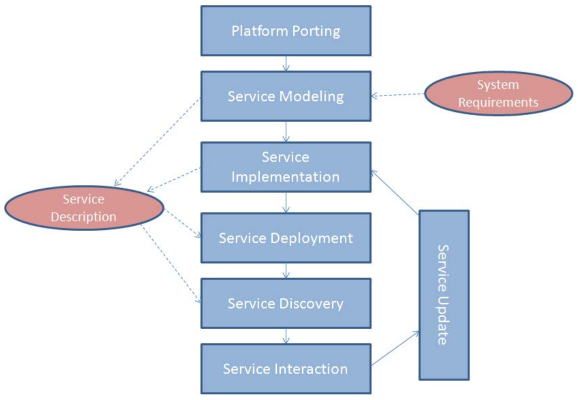

Fig. 2 Development steps of a smart grid application according to the proposed reference model for smart grid

4 Technological Implications of a Service-Oriented Reference Model for Smart Grid

As noted earlier in Section 2, the reference model for a domain helps in the process of developing a consistent set of standards and technologies for that domain. In this section, we present some technological implications of the proposed reference model for smart grid. In particular, we identify some technological requirements for enabling the development of smart grid applications according to the reference model, proposed in Section 3. We also present some potential solutions that can meet these requirements and are based on some state-of-the-art techniques from the domains of real-time systems and embedded control systems.

4.1 Technological Requirements

In order to enable the development of smart grid applications according to the proposed reference model, three major technological requirements are the following:

- A service description language.
- A service-based computing platform for smart grid computing nodes with support for resource-aware service deployment and QoS-aware service interaction.
- A service compiler

Figure 3 shows the role played by these technologies to enable the smart grid application development according to the proposed service-oriented reference model.

4.1.1 Service Description Language

According to the proposed reference model for smart grid, a service description plays a central role. Any smart grid application is modeled as a set of interacting services, each with its own service description. These service descriptions contain the following information:

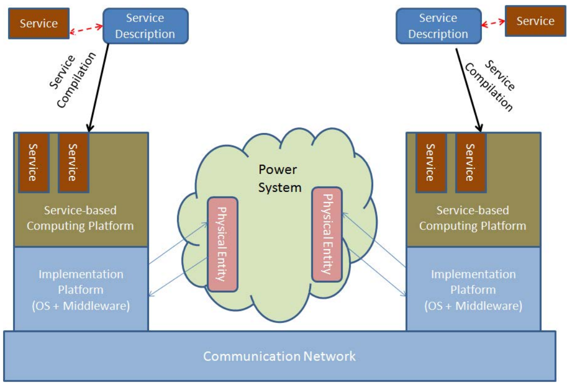

Fig. 3 Technological requirements of proposed smart grid reference model

Service Interface

The *service interface* section of a service description describes the messages that the service exchanges with other services and sensing and control actions that a service takes on the co-located physical entities. This section also identifies the QoS constraints on these messages and sensing and control actions.

Service Resources

The *service resources* section of a service description describes platform resource requirements of a service in order to satisfy the QoS constraints identified in the *service interface* section.

Service Modes

Unlike automotive and avionics systems, smart grid is a wide-area system. As a result, QoS constraints on message exchange among computing nodes of smart grid cannot be guaranteed by the communication subsystem. Therefore, service description for a service must contain a section which defines different modes of operation of the service for different QoS-fault scenarios.

In order to develop service descriptions that contain the above mentioned information (*service interface*, *service resources*, and *service modes*), an appropriate service description language (SDL) is required.

4.1.2 Service-Based Computing Platform for Smart Grid Computing Nodes

To enable the development of smart grid applications according to the proposed service-oriented reference model, each smart grid computing node must have an appropriate service-based computing platform that can support resource-aware service deployment and QoS-aware service interaction. A typical smart grid scenario involves a heterogeneous set of computing nodes with different processors, operating systems, and middleware technologies. Therefore, the required service-based computing platform must be capable of being ported onto these heterogeneous computing nodes.

4.1.3 Service Compiler

In order to properly deploy a service onto the service-based computing platform, an appropriate service compiler is required. This service compiler must be able to read the service description (specified using an appropriate service description language) and decide whether a certain computing node has enough resources to successfully deploy this service such that the service can meet its QoS constraints.

4.2 Potential Solution Approaches

In Section 4.1, a number of technological requirements have been identified that must be met before the proposed reference model can be utilized for the development of smart grid applications. Although the development of technologies that meet the identified requirements is a topic of ongoing research, in this section, we identify some potential solutions for each of these technological requirements. The proposed solutions are grounded in some state-of-the-art techniques, reported in the literature of embedded control systems and real-time systems.

Our proposed solutions are influenced heavily by the research on Giotto [6], a programming language for embedded control systems, and E Machine [5], a virtual machine that serves as the target for compilation of Giotto programs. Figure 4 shows the Giotto and E Machine configuration for a typical distributed real-time system. This configuration has been applied to local-area, distributed, real-time systems (such as automotive and avionics systems). We propose to extend this research for wide-area, distributed, real-time systems (such as smart grid), where QoS constraints on the message exchange cannot be guaranteed by the communication subsystem. In particular, we propose to extend Giotto programming language into the required *service description language*, E Machine into the required *service-based computing platform*, and Giotto compiler into the required *service compiler*. Figure 5 shows the technological requirements (originally shown in Figure 3) with the proposed solution approaches.

4.2.1 Service Description Language

A Short Review of Giotto Programming Language

The typical development process for an embedded control system can be divided into two steps: *control design* and *software implementation*. During the *control design* phase, a control engineer models the plant behavior and disturbances, derives the feedback control laws, and validates the performance of plant under the

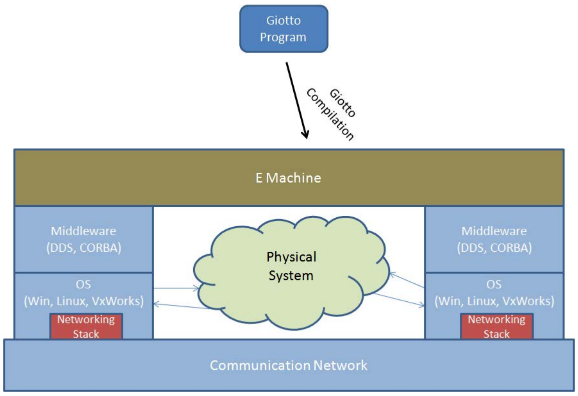

Fig. 4 Typical configuration of Giotto and E Machine for embedded control systems

influence of feedback controller through mathematical analysis and simulations. During the *software implementation* phase, a software engineer breaks down the feedback controller's computational activities into tasks and associated timing constraints on the completion of these tasks. Then, the software engineer develops code for these tasks in a traditional programming language (such as C) and assigns priorities to these tasks so that the tasks could meet their timing constraints while being scheduled on a processor by the scheduler of a real-time operating system (RTOS).

Giotto programming language aims to bridge the communication gap between control engineer and software engineer by providing an intermediate level of abstraction between control design and software implementation [6]. Giotto language syntax can be used by a Giotto program to specify time-triggered sensor readings, actuator updates, task invocations, and mode transitions. Then, a Giotto compiler must be used to compile (an entirely platform independent) Giotto program onto a specific computing platform. The compiler must preserve the functionality as well as the timing behavior specified by the Giotto program.

Figure 6 shows the major elements of Giotto syntax: *task*, *mode*, *driver*, *port*, and *guard*. *Task* is the basic functional unit of Giotto language and represents a periodically executable piece of code. Giotto *tasks* communicate with each other as well as with sensors and actuators. However, in Giotto, all data communication occurs through *ports*. In a Giotto program, there are mutually disjoint sets of task ports, sensor ports, and actuator ports. Task ports are further divided into task input ports, task output ports, and task private ports. Each *task* also has an associated function f (implemented in any sequential programming language) from its input ports and private ports to its output ports and private ports. According to Giotto semantics, sensor ports are updated by the environment while task ports and actuator ports are updated by the Giotto program.

Driver represents a piece of code that transports values between two *ports*. A *driver* can also have an associated *guard*, which is some boolean-valued function on the current values of certain *ports*. The code associated with the *driver* only executes if the *guard* of the *driver* evaluates to *true*. According to Giotto semantics, a *task* is an application-level code that consumes non-negligible amount of CPU

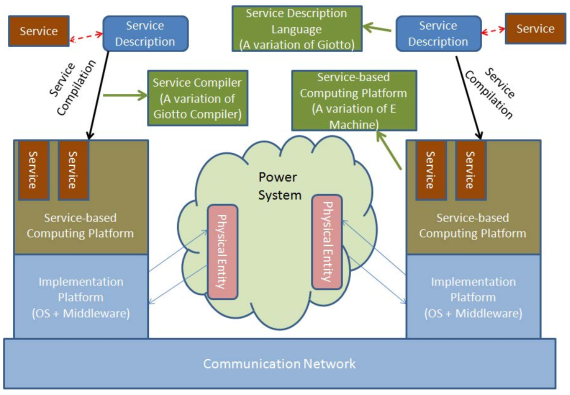
Service
Service Description
Service Description Language (A variation of Giotto)
Service Description
Service
Service Compilation
Service Compiler (A variation of Giotto Compiler)
Service-based Computing Platform (A variation of E Machine)
Service Compilation
Service
Service
Service-based Computing Platform
Power System
Physical Entity
Physical Entity
Service
Service
Service-based Computing Platform
Implementation Platform (OS + Middleware)
Implementation Platform (OS + Middleware)
Communication Network

Fig. 5 Technological requirements with potential solution approaches

time, while *driver* is a system-level code that can be executed instantaneously before the environment changes its state.

At the highest level of abstraction, a Giotto program is essentially a set of *modes*. At a certain instant of time, Giotto program can only be in one of its *modes*. However, during its execution, a Giotto program transitions from one *mode* to another based on the values of different *ports*. These possible *mode* transitions are specified in Giotto syntax through *mode swithces*. A *mode switch* specifies a target *mode*, switch frequency, and a guarded *driver*. Formally, a Giotto *mode* is made up of several concurrent *tasks*, a set of *mode switches*, a set of mode *ports*, a set of actuator updates, and a period. Each *task* of a *mode* specifies its frequency of execution per *mode* period. While Giotto program is in a certain *mode*, it repeats the same pattern of *task* executions for each *mode* period.

Figure 6 shows a Giotto program with two *modes*, m1 and m2. *Mode* m1 has two *tasks*, t1 and t2, while *mode* m2 has only one *task*, t3. *Mode* m1 has a period of 10ms, while *mode* m2 has a period of 20ms. *Task* t1 has a frequency of 2, while *task* t2 has a frequency of 1. This means that as long as Giotto program is in *mode* m1, *task* t1 executes every 5ms while *task* t2 executes every 10ms. Moreover, in this example, there is a *mode switch* from *mode* m1 to *mode* m2 with a switch frequency of 2. This implies that the *mode switch* condition (provided by the *guard* of *driver* d5) is tested every 5ms.

Extension of Giotto as a Service Description Language

Although Giotto was originally proposed as a programming language for embedded control systems, it can also be used as the service description language, required by the proposed smart grid reference model, with certain extensions. As outlined in Section 4.1.1, a service description must specify *service interface*, *service resources*, and *service modes*. The current Giotto syntax is capable of specifying all these requirements, except for the input and output messages of a service and QoS

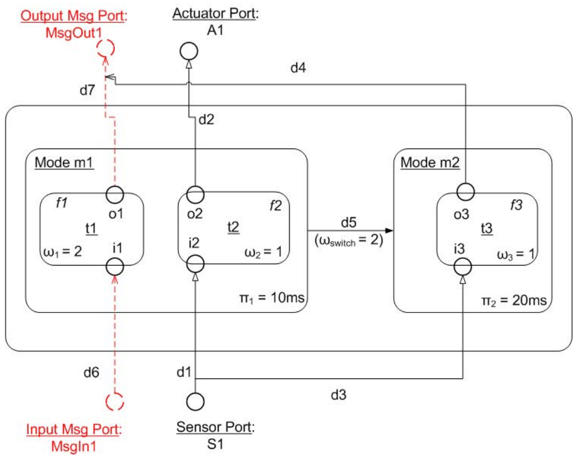

Fig. 6 Major programming elements of Giotto language. Proposed extensions are shown in red with dotted lines.

constraints associated with these messages. In order to overcome this deficiency, we propose to extend Giotto syntax with two new types of *ports*: *input message port* and *output message port*. We also propose to attach the following attributes with these new ports: *TimeSinceLastUpdate* and *DelayInLastUpdate*. These attributes could be used in the guard conditions, present in mode swithces. As a result, Giotto can be used to specify mode switches based on the violation of QoS constraints associated with message exchanges among services. Figure 6 also shows these proposed extensions to Giotto syntax.

4.2.2 Service-Based Computing Platform for Smart Grid Computing Nodes

A Short Review of E Machine

Earlier in this chapter, we have described Giotto, a platform-independent programming language for embedded control systems. In real-time systems literature, development of Giotto compilers for various computing platforms has been reported [6]. However, while developing these Giotto compilers, researchers have found it useful to have an intermediate language, which does not support the high-level concepts of Giotto but still provides a lower level platform-independent semantics for mediating between physical environment and software tasks [5]. The concept of such an intermediate language has evolved into *E code*. Moreover, in the literature, the term *Embedded Machine* or *E Machine* has been used for a virtual machine that interprets the *E code* [5].

The proposed *E code* essentially has the following three instructions:

1. *Call driver*
2. *Release task*
3. *Future E code*

In the *E Code* terminology, a *task* is a piece of application-level code, whose execution takes non-zero time. When invoked with its parameters, a *task* implements a computational activity and writes the results to *task* ports. On the other hand, a *driver* is a piece of system-level code that typically enables a communication activity. For example, a *driver* can provide sensor readings as arguments to a *task* or load *task* results from its ports to an actuator. It is assumed that the execution of a *driver* takes logically zero time.

Call driver instruction starts the execution of a *driver*. As the *driver* is supposed to execute in logically zero time, the *E Machine* waits until the driver completes execution before interpreting the next instruction of E code. *Release task* instruction hands off a task to the operating system. Typically, the task is put into the ready queue of the operating system. Scheduler of the operating system is not under the control of the *E Machine*. The scheduler may or may not be able to satisfy the real-time constraints of the *E code*. However, a compiler (which takes into account the platform resources) checks the time safety of *E code*, generated from a higher level language, such as Giotto. Such a compiler attempts to rule out any timing violations by knowing the worst-case execution time (WCET) of all the tasks and by applying the schedulability results available in the real-time systems literature.

Future E code instruction marks a block of *E code* for execution at some future time. This instruction has two parameters: a trigger and the address of the block of *E code*. The trigger is evaluated with every input event (such as clock, sensor, or task output) and the block of *E code* is executed as soon as the trigger evaluates to true.

E Machine as the Foundation of Service-Based Computing Platform

The smart grid reference model, proposed in this chapter, requires each smart grid computing node to support a computing platform that can support resource-aware service deployment and QoS-aware service monitoring. Since *E Machine*, summarized in the last section, supports resource-aware deployment and QoS-aware execution of Giotto programs, and we have already proposed a Giotto-based service description language in Section 4.2.1, it is natural to leverage *E Machine* as the foundation of required service-based computing platform. However, as noted in the last section, *E code* must be generated by an appropriate compiler to ensure time safety. Therefore, the required service-based computing platform must combine the *E Machine* with an appropriate service compiler that ensures resource-aware service deployment on *E Machine*. However, the service compiler code itself is not hard real-time in nature. Therefore, we propose a design of the service-based computing platform that combines two virtual machines: a hard real-time Embedded Machine (E Machine) and a soft real-time Compiler Machine (C Machine). E Machine executes the hard real-time service code and C Machine executes the soft real-time code for service compiler. The resulting service-based computing platform is shown in Figure 7.

4.2.3 Service Compiler

Giotto compilers, reported in the literature, typically work in two phases: *platform independent* phase and *platform dependent* phase. *Platform independent* phase

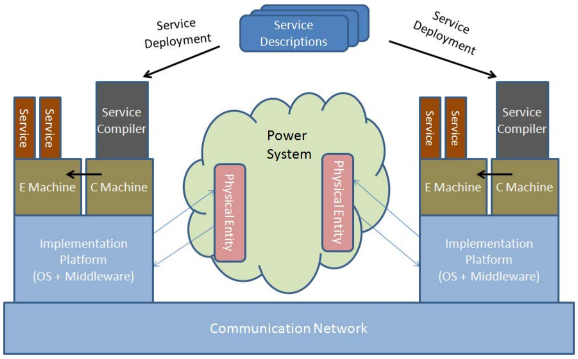

Fig. 7 Potential solution approach for the requirement of a service-based computing platform

generates *E code* from Giotto program; while *platform dependent* phase checks the time-safety of generated *E code* for a particular platform with known worst-case execution times and scheduling schemes. In the last two sections, we have proposed a service description language based on Giotto programming language and service-based computing platform based on *E Machine*. Therefore, it is possible to leverage the existing Giotto compilers and extend them into appropriate service compilers that can ensure resource-aware service deployment of services onto heterogeneous smart grid computing nodes.

5 Case Study: Demand Response

In this section, we present a case study of the application of the proposed reference model to a canonical smart grid application, demand response [3]. Figure 8 shows the demand response scenario under consideration. The power system topology for

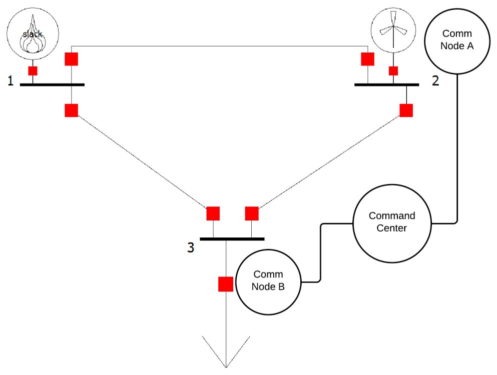

Fig. 8 System topology for the demand response scenario

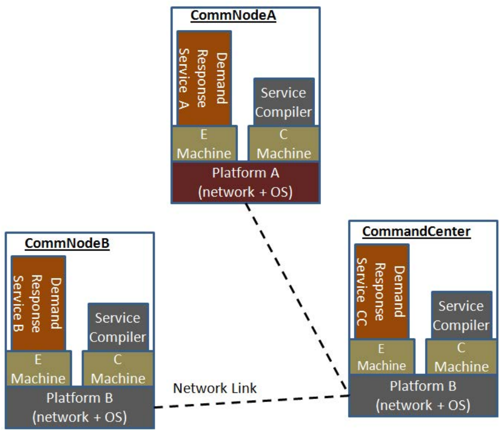
CommNodeA
Demand Response Service A
Service Compiler
E Machine
C Machine
Platform A (network + OS)
CommNodeB
Demand Response Service B
Service Compiler
E Machine
C Machine
Platform B (network + OS)
CommandCenter
Demand Response Service CC
Service Compiler
E Machine
C Machine
Platform B (network + OS)
Network Link

Fig. 9 Demand response case study (with proposed solution approach)

the demand response scenario consists of a wind generator at Bus2, an elastic load at Bus3 that tries to follow the intermittent wind generation at Bus2, and a gas generator at Bus1 that provides the slack. The communication system topology for this demand response scenario consists of three computing nodes: CommNodeA (co-located with wind generator), Command Center, and CommNodeB (co-located with elastic load).

According to the smart grid reference model, proposed in this chapter and shown in Figure 1, this demand response scenario can be described by an *application model*, a *platform model*, and a set of algorithms that facilitate application deployment on the platform. The *application model* for this demand response scenario consists of three services: *DemandResponseServiceA*, *DemandResponseServiceB*, and *DemandResponseServiceCC*. The *platform model* for this demand response scenario consists of three computing nodes (CommNodeA, CommNodeB, ControlCenter), one sensor (co-located with CommNodeA), one actuator (co-located with CommNodeB), three buses, three branches, two generators, and one load. Moreover, the application is installed on this platform by deploying *DemandResponseServiceA* on CommNodeA, *DemandResponseServiceB* on CommNodeB, and *DemandResponseServiceCC* on CommandCenter.

Figure 2 had shown the development steps of a smart grid application according to the proposed smart grid reference model. For the demand resposnse case study, presented in this section, the *platform porting* step consists of installing an appropriate service-based computing platform on the three computing nodes involved: CommNodeA, CommNodeB, and CommandCenter. The *service modeling* step consists of decomposing the demand response application into three services: *DemandResponseServiceA*, *DemandResponseServiceB*, and *DemandResponseServiceCC*. The *service implementation* step consists of developing service descriptions for these three services in an appropriate service description language. The *service deployment* step consists of deploying the three services, *DemandResponseServiceA*, *DemandResponseServiceB*, and *DemandResponseServiceCC* on three computing nodes, CommNodeA, CommNodeB, and CommandCenter, respectively. The *service discovery* step consists of these three services establishing initial contacts (for example, by setting up the lower level transport sockets). During the *service interaction* phase, *DemandResponseServiceA* reads from the co-located sensor and sends periodic messages to *DemandResponseServiceCC*, which in turn sends periodic messages to *DemandResponseServiceB*. The *service update* step will be required if we want to re-configure this demand response application (for example, by adding a new elastic load).

The rest of this section presents the application of proposed technological solutions to this case study. Figure 9 shows the cyber subsystem of demand response case study, where a service-based computing platform (consisting of a combination of E Machine and C Machine) has been ported onto each of the computing nodes and the appropriate service has been deployed on that computing platform through a service compiler. Table 1 shows the service description of *DemandResponseServiceB* using the proposed Giotto-based service description language, while Figure 10 shows the same service description graphically. *DemandResponseServiceB*

Table 1 Service Description of *DemandResponseServiceB* using the Proposed Giotto-based Service Description Language

```
Sensor Ports
    port customerOverride type binary
Actuator Ports
    port genPower type double
Input Message Ports
    port reqPower type double
Output Message Ports
    port status type binary
Task Input Ports
    port i1 type double
    port i2 type binary
Task Output Ports
    port o1 type double
    port o2 type binary
Task Private Ports

Tasks
    task t1 input i1 output o1 o2 function f1
    task t2 input i2 output o2 function f2

Drivers
    driver d1 source reqPower customerOverride
              guard g1 destination i1 i2 function h1
    driver d2 source o1 guard g2 destination genPower
              function h2
    driver d3 source o2 guard g3 destination status
              function h3
    driver d4 source customerOverride guard g4
              destination i2 o2 function h4
    driver d5 source customerOverride guard g5
              destination i2 function h5
    driver d6 source customerOverride guard g6
              destination i1 i2 o1 o2 function h6

Modes
    // Normal operating mode
    mode m1 period 10000ms ports i1 i2 o1 o2
      frequency 1 invoke task t1 driver d1
      frequency 1 update d2
      frequency 1 update d3
      frequency 1 switch m2 driver d4

    // User override mode
    mode m2 period 1000ms ports i2 o2
      frequency 1 invoke task t2 driver d5
      frequency 1 update d3
      frequency 2 switch m1 driver d6

Start m1
```

```
function f1( ) {
    o1 = i1;
    o2 = true;
}
function f2( ) {
    o2 = false;
}
function h1( ) {
    i1 = reqPower;
    i2 = customerOverride;
}
function h2( ) {
    genPower = o1;
}
...
...

binary guard g1( ) {
    return true;
}
...
...
binary guard g4( ) {
    return customerOverride;
}
binary guard g5( ) {
    return true;
}
binary guard g6( ) {
    return !customerOverride;
}
```

[a] Some guard and driver functions have been omitted to avoid unnecessary details.

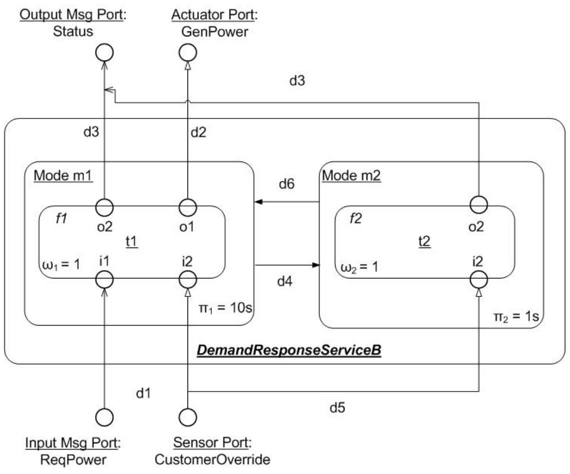

Fig. 10 Graphical representation of service descriptions for *DemandResponseServiceB*

consists of two *modes*: m1 (representing the normal operating mode) and m2 (representing the operating mode when the customer overrides the operation of demand response application). *Driver* d4 and *guard* g4 combine to describe the mode switch condition from m1 to m2, while *driver* d6 and *guard* g6 describe the mode switch condition from m2 to m1. Mode transitions between m1 and m2 occur based on the value of sensor port *customerOverride*, which represents the binary status of an application override user interface mechanism available to the customer. According to the service description, shown in Table 1, *mode* m1 has a period of 10000ms and it has a *mode switch* with the target *mode* of m2 and a frequency of 1, indicating that the mode switch condition is tested once every mode period. Therefore, mode switch condition from m1 (normal mode) to m2 (user override mode) is tested every 10 seconds.

6 Conclusion

Early smart grid applications have been developed in an ad-hoc manner without any underlying framework, resulting in a set of incompatible and inflexible smart grid technologies and standards. However, in the past, various engineering domains have successfully employed the concept of a reference model to enable clear communication among stakeholders and to serve as the underlying framework for the development of a consistent set of standards and technologies for that domain. In this chapter, we have presented an underlying reference model for smart grid that can enable the development of a set of compatible smart grid standards and technologies that are extensible to the future. The proposed reference model is based on some suitable extensions to the traditional service-oriented computing paradigm as this paradigm is uniquely suitable to handle the large scale, open nature, and long lifecycle of smart grid applications.

In this chapter, we have also identified some technological requirements (such as service description language, service-based computing platform, and service compiler) for enabling smart grid application development according to the proposed reference model. Although development of suitable technologies, which can meet these requirements, is a topic of ongoing research, we have presented potential solution approaches based on state-of-the-art techniques from real-time systems literature.

References

1. Brown, A., Carney, D., Feiler, P., et al.: A project support environment reference model. In: Proceedings of the ACM Conference on TRI-Ada, pp. 82–89 (1993)
2. Brown, A.W., Earl, A.N., McDermid, J.: Software engineering environments: automated support for software engineering. McGraw-Hill, New York (1992)
3. Conejo, A., Morales, J., Baringo, L.: Real-time demand response model. IEEE Transactions on Smart Grid 1(3), 236–242 (2010)
4. Erl, T.: Service-oriented architecture: concepts, technology, and design. Prentice Hall, New Jersey (2005)
5. Henzinger, T.A., Kirsch, C.M.: The embedded machine: predictable, portable real-time code. ACM Transactions on Programming Languages and Systems (TOPLAS) 29(6), 33 (2007)
6. Henzinger, T.A., Horowitz, B., Kirsch, C.M.: Giotto: A time-triggered language for embedded programming. Proceedings of the IEEE 91(1), 84–99 (2003)
7. Kopetz, H.: Real-time systems: design principles for distributed embedded applications. Springer, New York (2011)
8. Khaitan, S.K., McCalley, J.D.: Cyber physical system approach for design of power grids: a survey. In: Proceedings of the IEEE Power and Energy Society General Meeting, pp. 21–25 (2013)
9. Khaitan, S.K., McCalley, J.D.: Design techniques and applications of cyber physical systems: a survey. IEEE Systems Journal PP(99), 1–16 (2014)
10. Kundur, P.: Power system stability and control. McGraw-Hill, New York (1994)
11. Laplante, P.A., Ovaska, S.J.: Real-time systems design and analysis: tools for the practitioner. IEEE Press, New York (2012)
12. Liu, J.W.S.: Real-time systems. Prentice Hall, New Jersey (2000)
13. NIST Special Publication 1108R2, NIST Framework and Roadmap for Smart Grid Interoperability Standards, Release 2.0. (2012), `http://www.nist.gov/smartgrid/upload/NIST_Framework_Release_2-0_corr.pdf` (cited July 27, 2014)
14. OASIS Standard, OASIS Reference Model for Service Oriented Architecture (1999), `http://docs.oasis-open.org/soa-rm/v1.0/soa-rm.pdf` (cited May 21, 2014)
15. Regli, W.C., Mayk, I., Dugan, C.J., et al.: Development and specification of a reference model for agent-based systems. IEEE Transactions on Systems, Man, and Cybernetics, Part C: Applications and Reviews 39(5), 572–596 (2009)
16. World Economic Forum Report, Accelerating Successful Smart Grid Pilots (2010), `http://www.weforum.org/reports/accelerating-successful-smart-grid-pilots` (cited July 27, 2014)

17. Wolf, W.: Cyber-physical system. Computer 42(3), 88–89 (2009)
18. Wan, Y.: A Primer on Wind Power for Utility Applications. Technical Report: National Renewable Energy Laboratory (2005), `http://www.nrel.gov/docs/fy06osti/36230.pdf` (cited July 27, 2014)
19. Zimmermann, H.: OSI reference model–The ISO model of architecture for open systems interconnection. IEEE Transactions on Communications 28(4), 425–432 (1980)