Hindawi
Scientific Programming
Volume 2020, Article ID 3659849, 18 pages
https://doi.org/10.1155/2020/3659849

Research Article

INR: A Programming Model for Developing APPs of Insect Intelligent Building

Shuo Zhao,[1] **Qiliang Yang**,[1] **Jianchun Xing**,[1] **Qizhen Zhou**,[1] **Guangtong Xue**,[1,2] **and Wenjie Chen**[1]

[1]*College of Defense Engineering, Army Engineering University of PLA, Nanjing 210007, China*
[2]*78131 Troop of PLA, Chengdu 610000, China*

Correspondence should be addressed to Qiliang Yang; yql@893.com.cn

Received 16 September 2019; Revised 27 December 2019; Accepted 4 February 2020; Published 10 March 2020

Academic Editor: Basilio B. Fraguela

Insect Intelligent Building (I^2B) is a novel platform of intelligent buildings. The outstanding feature of I^2B is the decentralized network structure connected by smart nodes. I^2B can employ APPs (applications) developed by various practitioners or programming fans to manage and control buildings. However, due to the unique parallel operation of I^2B platform and the popularization of APP developers, there still exists no effective approach to supporting I^2B APP development. To deal with the challenges and provide meaningful guidance for describing and developing I^2B APP and motivating the prospective programming language design, we propose INR, a programming model for I^2B APP development. Three submodels in INR, namely, Individual, Neighborhood, and Region, are defined and implemented, respectively, for describing different task requirements. Moreover, new mechanisms of Tag-based programming and Clustering operation are established to support the plug-and-play and parallel abilities of APPs in I^2B. Finally, we apply the programming model into an application case to illustrate the developing pattern of the I^2B APP and verify the effectiveness of our approach.

1. Introduction

The Intelligent Insect Building (I^2B) [1–4] is a new type of intelligent buildings. Differing from traditional intelligent building control systems, I^2B adopts a decentralized network structure based on the space-distribution feature of buildings. The nodes of the decentralized network are smart nodes with computing capacity, called Computing Process Node (CPN). Each CPN corresponds to a building space unit or an electromechanical equipment. CPN contains a standard information model that can integrate and manage standardized building control information. All smart nodes in I^2B connect to its at most six spatial neighbors through its six data ports other than to a unified central processing node, which makes the whole building constitute a powerful decentralized network and achieve complex control tasks in a parallel and distributed way. Figure 1 shows the platform structure of I^2B. In addition to the CPN network hardware system, the I^2B platform has an open network community, i.e., the APP store. It includes all kinds of APPs enveloping building control strategies from the developers and can provide download services for building managers. I^2B has been developed as a new weather vane of intelligent buildings with the advantages of plug-and-play, efficient information sharing, self-organization, easy operation, easy expansion, etc. The core of achieving I^2B control is the APP, but there are some challenges for I^2B APP developing.

On the one hand, the APP oriented to decentralized, parallel, and distributed features of I^2B is the key to ensure the efficient and stable operation of I^2B. However, the new structure and features of the I^2B platform also present new difficulties and challenges for I^2B application development due to their profound changes. Furthermore, with the expansion of the openness and popularity of I^2B, the I^2B APP developers are no longer limited to building system engineers, but gradually expand to the public level including operation and maintenance managers, building users, etc. Therefore, how to orient to the I^2B features and public

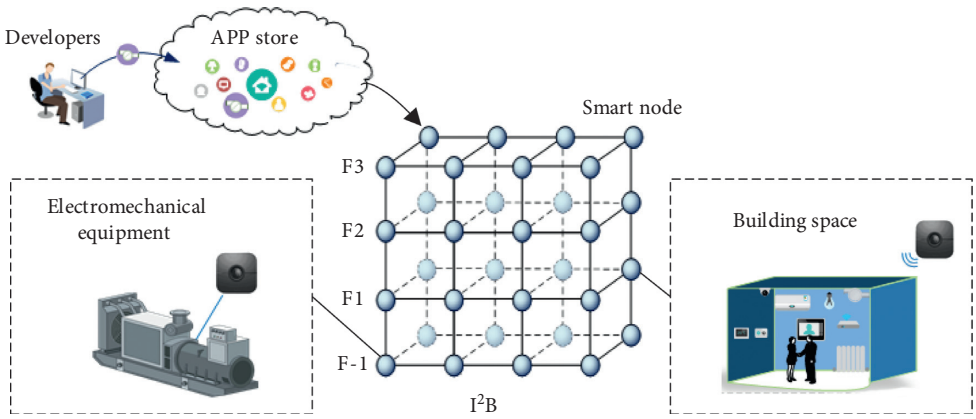


Figure 1: The platform structure of I²B.

developers and provide friendly, simple, and intuitive descriptions and methods of APP development has become a new requirement for I²B.

On the other hand, a programming language is a direct and effective solution for developing APPs. But General-Purpose Programming Languages (GPL) such as C language, Java, and Python tend to be difficult and error-prone for developers when facing the specific domain of I²B and cannot provide friendly and effective support due to the decentralized structure and parallel operating progress of I²B [5, 6]. A Domain-Specific Language (DSL) is a kind of specialized programming language designed for particular domain with simple and effective language components shaped by the characteristics of the domain [7, 8]. A DSL for I²B is an ideal approach to develop the I²B APPs. However, DSL designing and developing is a big project and complex process. Thus, how to guide the design and lead to an effective programming language to achieve programming of I²B control is a present challenge.

Domain modeling is the first step of solving a complex domain-specific problem [9]. It focuses on domain components, exploring key domain concepts and building relationships between the concepts, rather than cutting them into data and behavior and causing deviations in demands. It refines and updates the project to reflect the actual understanding of the "problem space" [10, 11].

Based on such considerations, in our previous work [12], we established a programming model named MSpro for I²B through summarizing the characteristics of master-slave distributed tasks in I²B and presenting the abstractions of them. The programming model constructed APPs by using new components such as *Domain*, *master CPN*, and *salve CPN*. Although it solves the problem of separating software development from building configuration, the MSpro programming model only focuses on one type of control tasks and cannot describe the entire I²B. Thus, it does not have enough describing and guiding significance for the APP development and the programming language design of I²B.

To effectively deal with the challenges of I²B APP description and development, this paper greatly extends the previous study [12] and proposes the I²B APP programming model named INR (i.e., *Individual*, *Neighborhood*, and *Region*) and its formal description to describe the different control tasks in I²B. The programming model is a systematic depiction of I²B APP development and can be a general guide for the prospective design of I²B programming language. The main contributions are as follows.

(1) We are the first, as far as we know, to establish the I²B APP programming model and its formal description.

(2) We propose and define three key programming concepts, i.e., *Individual*, *Neighborhood*, and *Region*, which also are three programming submodels that specifically describe and define the local task, network task, and the entire control task in I²B.

(3) We propose the novel mechanisms of Tag-based programming and Clustering operations to provide support for the features of I²B APP.

(4) We apply the programming model in an application case of the optimized operation of variable-air-volume air-conditioning systems to illustrate the developing pattern of the I²B APP and verify the effectiveness of our approach.

The rest of the paper is organized as follows. In Section 2, we introduce the related work. In Section 3, we present the INR programming model for developing I²B APP. Three programming submodels of *Individual*, *Neighborhood*, and *Region* are concretely discussed in Section 4. Then, in Section 5, we conducted an experiment that applied the INR model in an application case to introduce the developing pattern of the I²B APP and verify the effectiveness of our approach. Finally, we conclude the paper in Section 6.

2. Related Work

I²B APPs contain different control tasks, and to achieve these control tasks in I²B, many intelligent control algorithms based on its decentralized structure are proposed. Wang et al. [13] proposed a novel decentralized sensor fault detection and self-repair method for heating, ventilation, and air-conditioning systems. Yu et al. [14] presented a fully decentralized optimization algorithm based on log-linear model to complete pumps group operation, consisting of

many same-type pumps, under the least total power consumption. Wang and Zhao [15] proposed a distributed algorithm focusing on the building space topology matching problem. Zhang et al. [16] proposed a decentralized state estimation algorithm for building electrical distribution network. These achievements cover various control tasks in I^2B and follow the decentralized operating mechanism, but they cannot give general methods for I^2B development from a high level to synoptically describe the operation methods of I^2B.

The operation mechanism of I^2B is decided by its decentralized structure, which is similar to the parallel, distributed system and multiagent system (MAS). Some studies explored the models of development in these systems. Hossain et al. [17] presented μSETL, a programming abstraction for sensor networks based on set theory which can offer a powerful formalism and high expressiveness. Wang et al. [18] introduced a transformer programming framework including a model-specific system to facilitate the building of diverse data-parallel programming models. Zhang et al. [19] set forth the ThreadXML programming model, which is a newly rapid development multithreading programming model and can be embedded into most normal programming languages. Tekinerdogan and Arkin [20] presented ParDSL, a domain-specific language framework for providing explicit models to support the activities of mapping parallel algorithms to parallel computing platforms. Ferber and Gutknecht [21] presented a generic metamodel of multiagent systems called AALAADIN, based on organizational concepts to define a very simple description of coordination and negotiation schemes through multiagent systems. Hu et al. [22] proposed a novel AOP approach, namely, Oragent, for constructing and implementing dynamic and flexible systems. Though the models in these achievements can support parallel network developing, they take no consideration of the I^2B domain characteristics and cannot directly describe the I^2B control tasks. Besides, they mostly focus on the underlying implementation mechanisms so that the friendliness for users is not enough.

Current research of specific I^2B control tasks and the models of parallel, distributed systems has achieved remarkable results. However, the achievements only focus on specific fields with their own special features and do not support the intuitive and simple portrayal of I^2B features; thus, they cannot be directly applied to I^2B APP development. Therefore, there is currently no available method of I^2B APP development, and there is a lack of programming models to systematically describe I^2B APP development and generally guide the prospective design of I^2B programming language is requisite.

3. Overview of the INR Programming Model

The programming model is the pivotal structure for application development, as well as the foundation and core of the programming language. Programming languages of different paradigms adopt different programming models to map the actual problem models (located in the "problem space") to the machine model (located in the "solution space") [23]. Therefore, the programming model directly determines the difficulty of the programming language to solve the problems. Since traditional programming models are difficult to describe the I^2B control tasks in a simple and effective way, we propose the programming model for APP developing in the field of I^2B, which provides effective theoretical support for further language definition and grammar design of the I^2B programming language.

3.1. General Architecture of INR. In this subsection, we first present the formal definition of INR and then establish its general architecture.

Definition 1. Programming Model. A programming model can be formally defined as a two-tuple:

$$pmo = \langle P_A, P_{Mec} \rangle, \tag{1}$$

where P_A denotes a collection of programming abstractions. Programming abstractions $p_A \in P_A$ are used to describe the essential features of things and are the basic elements that make up a software entity. P_{Mec} denotes a collection of programming mechanisms. Programming mechanisms $p_{Mec} \in P_{Mec}$ are used to describe the interaction between programming abstractions P_A; that means $p_{Mec} = P_A \times P_A$.

In a process-oriented programming language (such as C, Fortran), using a one-dimensional *variable* programming abstraction to model things can only reflect the data characteristics of things, and the abstraction is of low level [24]; its programming mechanism is only used to describe the interactions between variables (such as assignments, comparisons, and function operations). Object-oriented languages (such as Java and C++) encapsulate *state* and *behavior* into the *object* programming abstraction, depicting the features of things from two dimensions and further improving the level of abstraction; its programming mechanism contains not only low-dimensional variables, but also the interaction between the objects, such as the combination and messaging of objects, which is closer to the problem space [25, 26]. The agent-oriented programming abstraction in agent-oriented programming languages introduces individual thinking attributes (belief, desire, and intention) and social attributes (organization, role, etc.) and becomes an autonomous, resident, social, and reactive high-dimensional unity with a higher level of abstraction and more complex programming mechanisms [27, 28].

For the development of I^2B APP and its programming language, the first task is to establish a programming model for the control task description and solution, which means using more effective programming abstractions and programming mechanisms to abstract the practical tasks to be solved in I^2B and to decompose the software entities in the applications. Since the control tasks of I^2B are complicated and diverse, this paper used the *separation of concerns* as the basic analysis principle. We decompose the programming features of I^2B APP development according to the local tasks and network computing activities [29] and propose the INR programming model. To make it intuitive and normative,

the proposed models are described with different abstract objects and UML notations of relations. The general architecture of INR is shown in Figure 2, which includes three submodels: *Individual*, *Neighborhood*, and *Region*.

Definition 2. INR Programming Model. It is the programming model for developing I^{2}B APPs including three submodels, which can be defined as a three-tuple:

$$INR = \langle Individual, Neighborhood, Region \rangle, \quad (2)$$

where *Individual*, *Neighborhood*, and *Region* are three submodels that are, respectively, oriented to different control tasks.

(1) The *Individual* programming submodel is oriented to the local task and provides programming abstractions such as *basic unit* (the specific definitions and introductions of the programming abstractions of submodels will be given, respectively, in the following pages) and *slave device* and their programming mechanisms, which can effectively characterize the internal state and individual behavior of the smart nodes to meet the programming requirements for local tasks.

(2) The *Neighborhood* programming submodel is oriented to the network computing activity, providing programming abstractions such as *self* and *neighbor* and their programming mechanisms. It can effectively describe the interaction behaviors between a smart node with its neighbor nodes to meet the programming requirements for decentralized network computing activities.

(3) The *Region* programming submodel is oriented to the entire control task and parallel operation, with the highest level of abstraction. It provides programming abstractions such as *region* and *basic unit* and their programming mechanisms. Through the dynamic binding mechanism of *region* and *basic unit*, the local task and the network computing activity can be effectively integrated, which meets the programming requirements of the entire control task.

Figure 2 shows that the three programming submodels are at different levels, but not completely parallel and independent. The *Individual* is a concrete implementation of the *basic unit* programming abstraction in the *Region*; the network computing activity in the *Region* is a further encapsulation of the *Neighborhood* programming. The INR programming model adopts the thought of separation of concerns and provides I^{2}B developers with important support for modular decomposition from the programming model level, which can realize effective and convenient description of the application tasks, thus improving the constructivity and packaging capabilities of APP development.

3.2. Key Mechanisms of INR. The INR programming model provides important domain features combining the programming requirements of the I^{2}B control tasks and key mechanisms to achieve such domain features as plug-and-play of APPs in I^{2}B. The proposed key mechanisms include Tag-based programming and Clustering operations.

3.2.1. Tag-Based Programming. Tag-based programming is based on tags to implement access to objects and calls. The core idea of Tag-based programming is to realize the decoupling between the logical address of the building abstraction in the APP program and the physical address of the actual building device entity, thus realizing the plug-and-play of APP. In other words, as shown in Figure 3, the APP developers set a tag during programming, and then they only need to focus on the tag during the development process, and the logical address of the label is managed by the tag manager correspondingly; when the APP is downloaded to the local, through a small-scale manual or automatic local configuration, the corresponding connection between physical addresses of the building device entities and the tag is established in the tag manager, thereby realizing control of the device entities by the tag. During the development and operation of the entire APP, the logical address and physical address are no longer consistent one-to-one correspondence. Through this approach, the developers do not have to consider the number and the physical addresses of all controlled objects every time. Inversely, they only need to pay attention to the abstract categories of controlled objects, which can add or change the objects it contains at any time according to the needs of the environment. Therefore, the programming process can get rid of the dependency on the node number and the communication address.

3.2.2. Clustering Operation. Clustering operation is based on Tag-based programming and refers to the tag-oriented function operation, which is automatically applied to all objects that own the corresponding tag. As shown in Figure 3, through the decoupling between the logical address and the physical address, the association of a single tag to multiple devices can be achieved. Coordinating with the Tag-based programming, it provides I^{2}B APP developers with a simple and fast development process that allows them to achieve operation and control of a group with only some simple codes. By encapsulating a series of complex control methods, Clustering operations can implement large-scale operations with simple statements.

These key mechanisms run through the entire programming model and they have relatively different functions and usages in different submodels, which will be specifically introduced in the different submodels hereinbelow.

4. Implementation of INR

In this section, we present the implementation of INR. In other words, the realization of the three submodels in INR, i.e., *Individual*, *Neighborhood*, and *Region*, is explored.

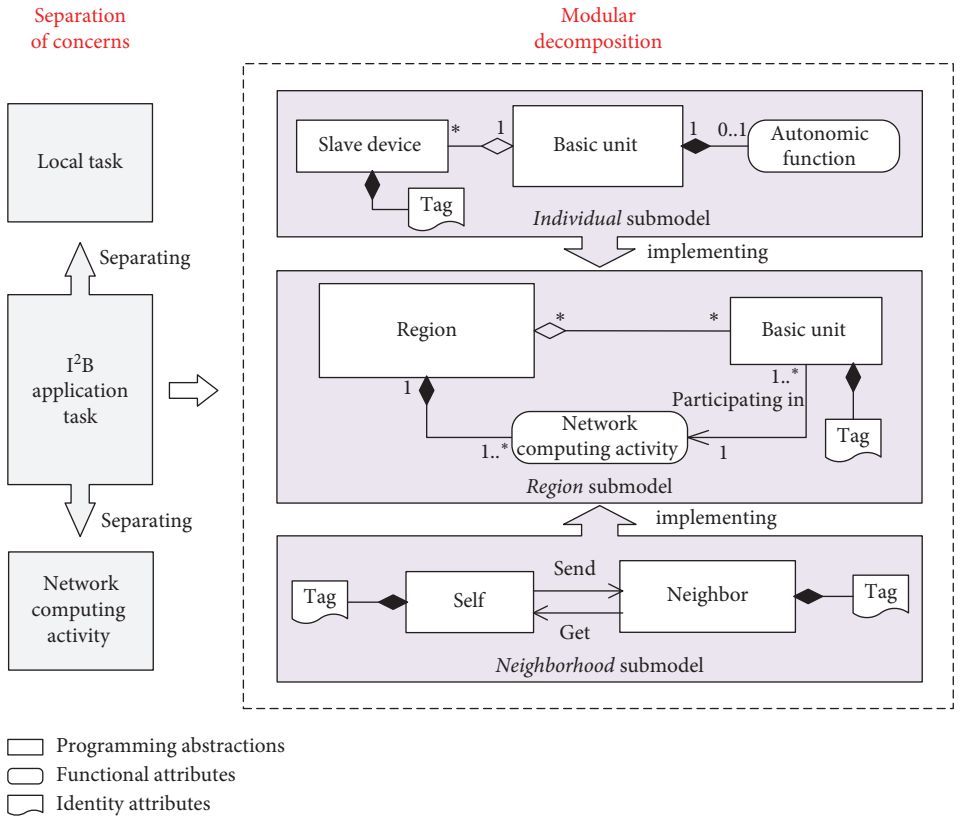


Figure 2: General architecture of INR.

4.1. Defining Individual in INR. This section first analyzes the programming requirements of local tasks, then proposes the *Individual* programming submodel, and introduces the related concepts, abstractions, and programming mechanisms in detail.

4.1.1. Problem Analysis. Local tasks (also known as single-point tasks) are internal operational management tasks that are independently performed by the smart nodes, such as the perception of the building environment area, the monitoring of internal operating conditions, the regulation of parameters, the control of slave devices, and the management of information. The following describes the programming requirements of local tasks in combination with a specific application case.

Case 1. Implementing the control strategy that the number of people controls the lights: when there is someone in the room, turn on all lighting; otherwise, turn off all lighting (assume that there are three lights in the room; the numbers of slave devices are No. 1, No. 2, and No. 3).

Figure 4 shows the pseudocode to implement this case based on the traditional programming paradigm, where "Light_1·SwicthSet" represents the switch setting parameter of the No. 1 lighting device in the smart node information model. In this procedure, the control of the lighting device is dependent on the lighting device number, resulting in poor program portability. In other words, if the program is downloaded to other rooms, the function can be guaranteed only if the lighting devices' numbers of the procedure and the rooms are identical. However, due to the large differences in different rooms, it is difficult to ensure that the numbers of slave devices are identical. Therefore, to meet the generalization requirements of I^2B control task, it is necessary to get rid of the dependency on the slave device number during programming. Moreover, for the control of slave devices, the application needs to have great flexibility; that is, in different application scenarios, the same program can be applied to a specific slave device as well as to a collection of the same-type devices to perform group management. For example, for the above application case, a more flexible function is that the application can choose one or more lighting devices to control according to the personalized demands of different rooms.

Since there are different types of smart nodes in I^2B, the application of the local task must be executed in the matching smart nodes (for example, the program of number of people controls the lights can only run in the building space units). Therefore, an APP that involves a local task must indicate the type of the smart node.

Moreover, due to environmental changes, human factors, and their own control strategies, local tasks often need to be triggered by timing or events. For example, if the execution of the above case program is triggered by the change of the people number, the CPN computing resources can be fully and efficiently utilized and the reactivity of program can be improved. Moreover, when there are

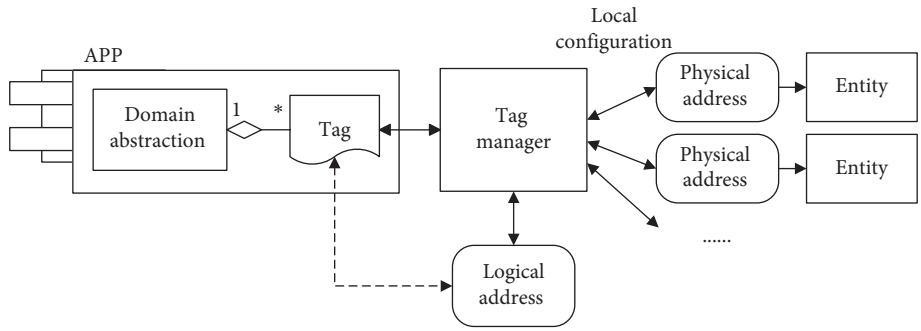


Figure 3: Implementation concept of Tag-based programming.

```
if (number==0){                    //if the number of people is 0
        Light1_SwicthSet=0;  //then turn off the light 1
        Light2_SwicthSet=0;  //and turn off the light 2
        Light3_SwicthSet=0;  //and turn off the light 3
        }
else {                             //if the number of people is more than 0
        Light1_SwicthSet=1;  //then turn on the light 1
        Light2_SwicthSet=1;  //and turn on the light 2
        Light3_SwicthSet=1;  //and turn on the light 3
        }
```

Figure 4: The pseudocode of implementing the control strategy that the number of people controls the lights, based on the traditional programming paradigm.

multiple relatively independent tasks in the application, the tasks need to be processed in parallel.

In addition, local tasks are independently and autonomously performed by a single smart node and do not need to interact with other smart nodes, which also needs to be reflected in the programming model.

The programming requirements for local tasks are summarized below:

(i) The flexibility of slave device control and independence of device numbers

(ii) Association with the smart node types

(iii) Parallel and triggered execution

(iv) Independence and autonomy

4.1.2. Individual Programming Submodel. In the I^{2}B physics world, the smart node is an individual with independent autonomous operation capability (such as building space unit and intelligent equipment unit, including water pumps, AHU (Air Handle Unit), etc.), and the local task can be regarded as that the smart node controls and manages its own resources. Inside the smart node, various information parameters and slave devices are contained. The slave device is a special type of internal member (such as lighting equipment and smart curtains in a building space unit) with relatively independent data attributes and functional attributes, but it cannot operate autonomously and still needs to be managed by the smart node.

According to the programming requirements of local tasks and based on the operating mechanism of the I^{2}B in the physical world, this section proposes the *Individual* programming submodel, as shown in Figure 5. The model uses programming abstractions, status attributes, functional attributes, and identity attributes to represent different abstract objects and UML relationships to describe their relationships. Programming abstraction is the basic element to depict the control object in application tasks, and status attribute, functional attribute, and identity attribute are used to describe the characteristics of the programming abstractions, which represent the structure of required data, functional effect, and identifying features. This programming submodel introduces two basic programming abstractions of *basic unit* and *slave device*. A *slave device* is the internal member that belongs to a *basic unit*, and the *basic unit* is responsible for managing the *slave device*, which is consistent with the physical meaning of reality. Local tasks in I^{2}B can be modeled and described by the *basic unit* programming abstraction and the interaction mechanisms between its internal members. The main programming abstractions and programming mechanisms involved in the *Individual* submodel are described in detail as follows.

(1) Programming Abstractions in Individual

Definition 3. Basic Unit. The basic unit is an abstraction of the smart node which corresponds to a building space unit or an electromechanical equipment. Basic unit can be defined as a six-tuple:

$$unit = \langle name, type, D, f_{auto}, F_{static}, Dev_{sla} \rangle, \quad (3)$$

where *name* indicates the name of the *basic unit*. It is an identifier used to distinguish it from other *basic units*. It is unique in the same APP and needs to be specified by humans. *type* denotes the type of the *basic unit*, which is used to characterize the type attribute of the *basic unit*. It is designed for the local task requirements associated with the *basic unit* types. D represents a collection of *basic unit* data members used to characterize the state attributes of the *basic unit*. F_{static} represents a collection of *basic unit* static functions used to characterize the static functional properties of the *basic unit*. Dev_{sla} denotes a collection of slave devices in *basic unit*. Slave devices $d_{sla} \in Dev_{sla}$ are relatively independent members in the *basic unit* (described in more

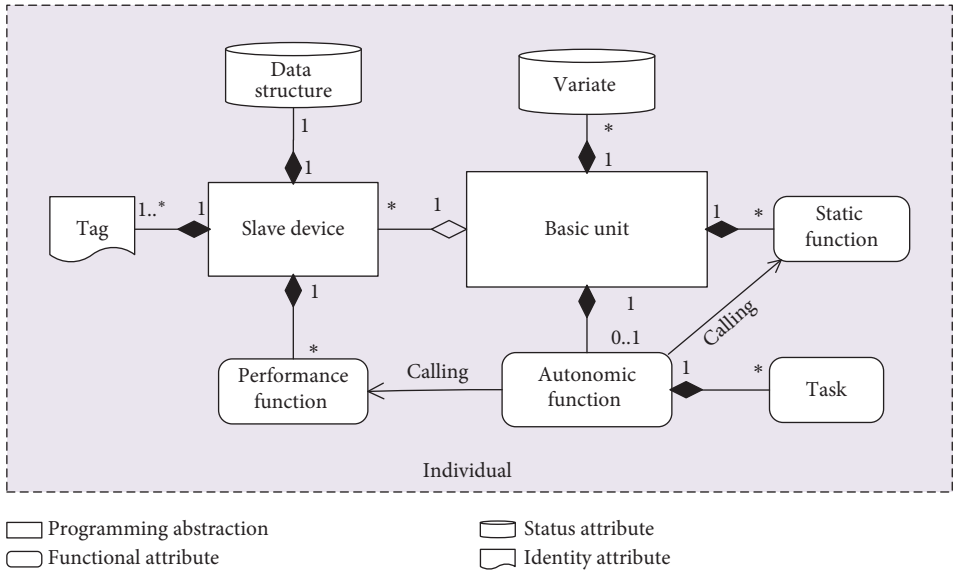


Figure 5: The Individual programming submodel.

detail below). f_{auto} denotes the autonomous function of the *basic unit*. It is unique and is used to characterize the dynamic behavior of the *basic unit*. It can autonomously call static functions or manage slave devices, embodying the independence and autonomy of the *basic unit*. Autonomous function contains a special class of dynamic behaviors—tasks T. $T = T_{non} \cup T_{trig}$, where T_{non} represents a collection of nontriggered tasks and T_{trig} represents a collection of triggered tasks. Tasks need to be created in autonomous function, and the relationship between tasks is parallel.

Definition 4. Slave Device. A slave device is an abstraction of the physical entity of a slave device in the building basic unit. It is also an internal member of the basic unit programming abstraction and can be defined as a four-tuple:

$$d_{sla} = \langle type, Tag, D_{str}, F_{sla} \rangle, \quad (4)$$

where *type* indicates the type of the slave device and is unique to the slave device. *Tag* denotes a collection of slave device tags. $tag \in Tag$, *tag* is used to replace the numbers of slave devices, as defined by a developer during the APP development phase. D_{str} indicates the data structure of the slave device, which is used to describe the state attribute of the slave device. The internal data $d_{str} \in D_{str}$ is derived from the *basic unit* information model, but is independent of the specific device number. For example, the switch setting value of the lighting device $d^{Light}_{SwitchSet}$ can be used as a data member in the data structure of the slave device, but does not specifically refer to the switch setting of any number of lighting devices. d_{str} can be accessed in the form of a slave device tag. F_{sla} denotes a collection of slave device performance functions, characterizing the functional properties of the slave device, which can be called by the static function, the autonomous function, and tasks in the form of a slave device tag.

(2) Task-Based Parallel Programming and Triggered Mechanism. As an important component of the autonomous function f_{auto}, task T is an important approach to implement the parallel and triggered execution of the local task program in the *Individual* programming submodel. In the execution process of autonomous function, tasks T have independent execution flows that can be used to implement parallel processing of local tasks. Moreover, according to the trigger attribute of task T, the triggered task and the nontriggered task can be further divided [30]. Figure 6 shows the main difference between the two types of tasks.

After the nontriggered task T_{non} is created in the main execution path of the autonomous function f_{auto}, it starts execution immediately and will be automatically destroyed after the execution is completed. The next execution still needs an autonomous function f_{auto} to make an explicit creation.

After the triggered task T_{trig} is created in the main execution path of the autonomous function f_{auto}, the execution does not start immediately, but waits for the arrival of the trigger condition; that is, when the trigger condition is satisfied, the execution is started; otherwise, it will be in a blocked state. After T_{trig} execution is completed, it is not automatically destroyed but waits for the next trigger, and the destruction needs to be clarified in the program. The programming mechanism of the triggered task can be used to implement the triggered execution of the applications.

(3) Tag-Based Programming and Clustering Operations Mechanisms. In the *Individual* programming submodel, the programming mechanism of Tag-based programming and Clustering operations based on slave devices is proposed to realize the flexibility of device control and independence of device numbers.

Tag-based programming refers to the invocation of access to a slave device data structure D_{str} and performance function F_{sla} based on slave device tags.

Clustering operations refer to functional operations based on slave device tags and the operations are

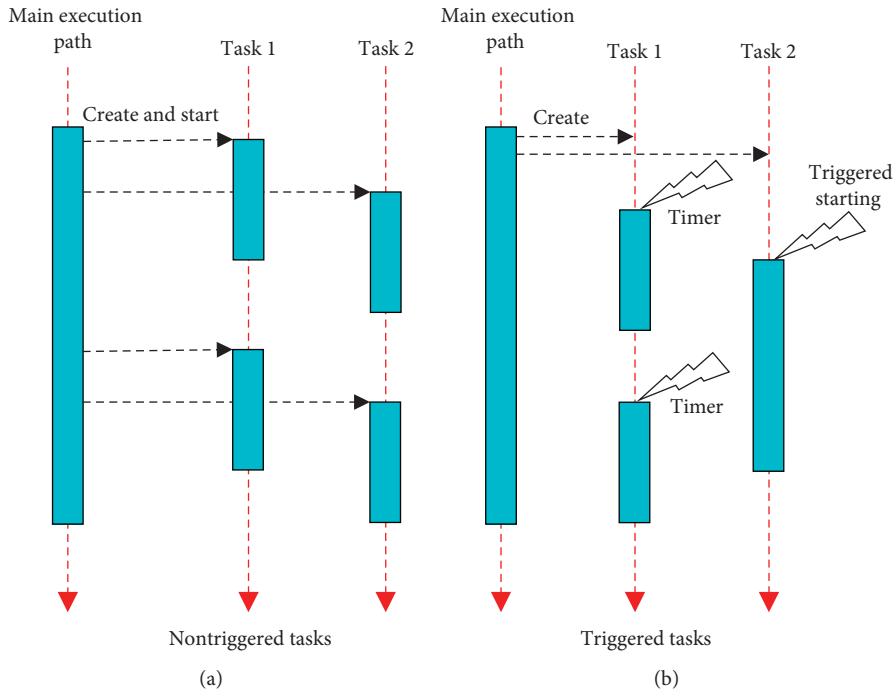


Figure 6: Two types of the task.

automatically applied to all slave devices that own the corresponding tag.

Specifically, when an autonomous function f_{auto} calls a slave device performance function F_{sla} (or accesses data d_{str}) based on a tag (e.g., tag_x), all tagged slave devices will execute the performance function (or access data d_{str}). The users can *stick* the corresponding tag to the slave device that needs to perform such operation during the APP installation or operation and maintenance phase as needed so that the dependency on the slave device number is got rid of and the flexibility of the slave device control is improved.

Table 1 shows the fundamental approaches by which the established *Individual* programming submodel meets the requirements of local task programming. (a) Satisfying programming requirement of dependency with *basic unit* types for local tasks by encapsulating their type attributes in the *basic unit* programming abstraction. (b) Implementing the control flexibility and number independence of slave device based on the Tag-based programming and Clustering operations of the slave devices. (c) Encapsulating autonomous functions for "*basic units*" to have "independence and autonomy" of behavior. (d) Adding *task* to autonomous functions to satisfy the programming requirement of the local task parallel and triggered execution.

4.2. Defining Neighborhood in INR. This section takes the decentralized network summation calculation as an example, analyzes the programming requirements of the I^{2}B network computing activities, then proposes the *Neighborhood* programming submodel, and introduces the relevant programming abstractions and programming mechanisms in detail.

4.2.1. Problem Analysis

Case 2. In I^{2}B, the decentralized network summation calculation is a typical control task based on the needs to be based on the decentralized network of I^{2}B. The summation calculation needs a tree-shaped computing network, as shown in Figure 6. During the whole calculation process, each node only interacts with adjacent nodes. Assuming that Q is a variable that needs to be summed, the specific execution logic is as follows:

(i) If it is a leaf node, the node passes its own variable Q to its father node.

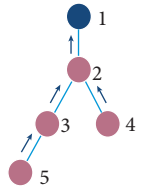
(ii) If it is an inner node, the node locally sums the Q_i passed by its child nodes with its own Q and then passes the sum result Q to its father node.

(iii) If it is a root node, the node locally sums the Q_i passed by its child nodes with its own Q, and the final result Q_{sum} is obtained.

When using the traditional parallel programming model (e.g., the message-transfer parallel programming model [31]) to program the decentralized network summation calculation, it is necessary to explicitly specify the node number or communication address of the interactive nodes in the program. The pseudocode of implementing the network summation based on the traditional parallel programming model is shown in Figure 7. Obviously, the programmed program is tightly coupled with the node numbers, causing the program to fail to meet the requirements of the generalization of the I^{2}B APP, which means that once the amount and numbers of nodes in the network change, the function of network summation cannot be realized through the same

Table 1: The fundamental approaches of Individual programming submodel for local task.

Programming requirements for local tasks	Approach
Dependency with *basic unit* types	Type attribute of the *basic unit*
Flexibility of the slave device control and independence of the device number	Tag-based programming and clustering operations of the slave devices
Independence and autonomy	Autonomous function of the *basic unit*
Parallel and triggered execution	Task-based parallel programming and triggered mechanism




```
NetSum_Q()
{
  if (N5) {                                  //if Node5
        send(Q,N3); }                        //send Q to Node3
  if (N4) {                                  //if Node4
        send(Q,N2); }                        //send Q to Node2
  if (N3) {                                  //if Node3
        Q=rev(Q,N5)+Q;                       //receive data from Node5 and sum with local data
        send(Q,N2); }                        //send Q to Node2
  if (N2) {                                  //if Node2
        Q=rev(Q,N3)+rev(Q,N4)+Q;             //receive data from Node5 and Node4 and sum with local data
        send(Q,N1); }                        //send Q to Node1
  if (N1) {                                  //if Node1
        Qsum=rev(Q,N2);                      //receive data from Node5
        return Qsum; }                       //rerturn the result
}
```

Figure 7: The pseudocode of implementing the network summation based on the traditional parallel programming model.

process. Therefore, the core of network computing activity programming is to get rid of the dependency of the node numbers (or communication addresses) when the nodes interact at the programming model level.

4.2.2. Neighborhood Programming Submodel. This section presents the *Neighborhood* programming submodel with network tags and interaction variables. As Figure 8 shows, this submodel includes two programming abstractions of *self* and *neighbor* and their status attributes, functional attributes, and identity attributes. There is a kind of special abstract object, connotative abstraction, which means that it is not used in the description process, but is the actual affected object. The model only focuses on two abstract concepts of *self* and *neighbor* when describing nodes' operation, and all nodes are self-centered without considering how many and who the neighbors are, because for each node, all their neighbors are a whole concept *neighbor*.

(1) Programming Abstractions in Neighborhood

Definition 5. Self. The self refers to an abstraction of the basic unit itself that participates in the network computing activity and can be defined as a four-tuple:

$$self = \langle num, V_I, T_{self}, O_N \rangle, \tag{5}$$

where *num* represents the node number of *self*, which is used as an identification to distinguish the self and others and is unique. The node number is the basis for establishing the neighborhood relationship. Although the programming model emphasizes hiding the node number information during the programming phase, node number support is still required during the network establishing phase. V_I denotes a collection of interaction variables. The nature of the interaction variable is a vector or array, i.e., $v_i = \{x_1, x_2, \ldots, x_n\} \in V_I$, where x_n is the data sent by the nth neighboring node. T_{self} denotes a collection of self-tags, characterizing the role that the node plays in the network. Different from the node number, the self-tag is not unique. This means that the same node can have multiple self-tags at the same time, and different self-tags can be used to represent one node in different application scenarios. O_N denotes a collection of neighborhood operations for the processing of interaction variables. The neighborhood interaction operator $O_{N-I} = \{send, get\} \subset O_N$ is a special kind of neighborhood operation, which is used to describe the interaction between the *self* node and its *neighbors*. Specifically, the operator *send* represents sending the data of *self* to the *neighbors*, and the operator *get* represents getting the data from the *neighbors*.

Definition 6. Neighbor. The neighbor is an abstraction of the collection of basic units in the adjacent relationship with self and can be defined as a three-tuple:

$$neighbor = \langle Num, T_{neigh}, R_n \rangle, \tag{6}$$

where *Num* represents a collection of node numbers of the neighboring *basic units*, hidden from the APP developers. T_{neigh} denotes a collection of *neighbor-tag*s, used to characterize the neighborhood relationship between *self basic unit* and neighboring nodes. R_n indicates the mapping relationship between the *neighbor-tag*s and the node numbers

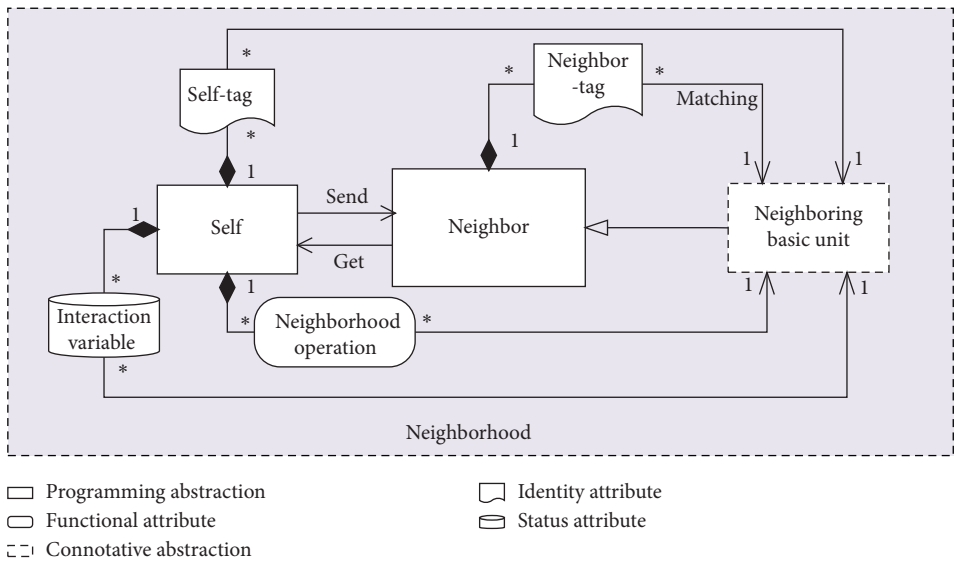


Figure 8: The Neighborhood programming submodel.

of adjacent *basic units*, i.e., $R_n \subseteq Num \times T_{neigh}$, which is hidden from the APP developers and maintained by the operating system.

Self refers to an abstraction of the *basic unit* itself that participates in the network computing activity, and *neighbor* is an abstraction of the collection of *basic units* in the adjacent relationship with *self*. *Self* and *neighbor* are relative to each other, meaning that each *basic unit* is its own self and also belongs to the neighbors of other adjacent *basic units*.

(2) Network Tags of Self and Neighbor. In the *Individual* programming submodel, the *slave device* programming abstraction with the Tag-based programming removes the dependency on the slave device number and solves the universality problem of the local task application. Analogously, the key to realizing the universality of network computing activity applications is to find suitable abstractions to replace the node numbers.

In the decentralized network computing activity, the node number mainly plays the role of characterizing neighborhood relationship and network role. Concretely, the so-called neighborhood relationship refers to the relationship between one node and its neighboring nodes. There may be different neighborhood relationships between the same node and different neighboring nodes. For example, as shown in Figure 6, for node 3, node 5 is its father node, while node 2 is its child node. The so-called network role refers to the role of nodes in the entire network. For example, as shown in Figure 6, the role of node 1 in the network is the root node, which only needs to receive the messages from the child nodes in the network summation calculation, while node 2 is the inner node, which needs to receive the data from the child nodes as well as send its own data to its father node.

Therefore, in order to replace the node number in network computing activity, this paper proposes two types of network tags: self and neighbor.

Definition 7. Self-Tag. The self-tag refers to a collection of network roles that a node plays in the I^{2}B network task.

Definition 8. Neighbor-Tag. The neighbor-tag refers to the collection of neighborhood relationships between the node participating in the I^{2}B network task and its neighboring nodes.

Take Case 2 as an example. The *self-tag* contains three tags: *root*, *leaf*, and *insider*, which are used to represent the three kinds of network roles: root node, leaf node, and inner node. The *neighbor-tag* contains tags of *child* and *father* to represent the information transfer relationship in the neighborhood. Based on the *self-tag* and *neighbor-tag*, the interaction between nodes in the neighborhood can be clearly described. For example, the statement *if (N5) then send (Q, N3)* in Figure 7 can be replaced with *if (self-tag is "leaf") then send (Q, "father")*. The latter is more versatile in the semantics expressed and can observably reduce the amount of code.

(3) Neighborhood Interaction Mechanism and Interaction Variables. Network computing activities are accomplished through information interaction between nodes. Therefore, it is necessary to provide interaction mode and technical means that adapt to I^{2}B network systems at the programming model level. As shown in Figure 9, common information exchange methods for distributed parallel computing networks include message passing and the mailbox method [32].

The message passing method needs to establish a one-to-one correspondence between nodes to achieve two-way peer-to-peer communication, as shown in Figure 9(a). Both parties of the interaction must perform the explicit receive/send operation to obtain the information of the other. This means that when one node sends out the message, the interaction will not be completed until the other node performs the receiving operation. In I^{2}B, there may be hundreds of nodes participating in a network task; thus, this mutual

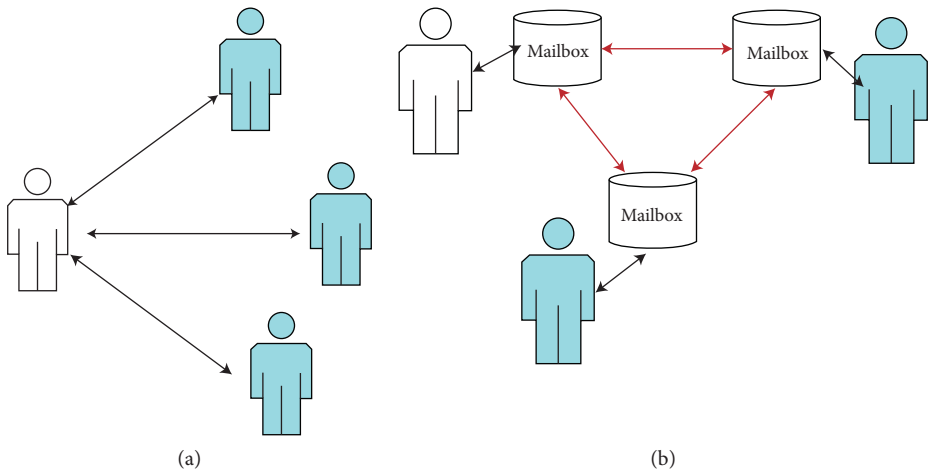


Figure 9: Information exchange methods for distributed parallel computing networks. (a) Message passing. (b) Mailbox.

waiting mechanism will seriously reduce the computing efficiency.

The mailbox method is shown in Figure 9(b), and as its name suggests, its interactive mode is similar to the e-mail used in human society. In the mailbox method, each node is required to have one *mailbox*, and the information is packaged into an *e-mail* and sent to the *mailbox* of the target node. Moreover, the node processes the information by accessing its own *mailbox*. Compared with the message passing, the mailbox method does not require the communication parties to wait synchronously and only needs to obtain the corresponding information in the mailbox when necessary. This method has good asynchrony and parallelism, so it is more suitable for I^2B.

In I^2B, the node only needs to communicate with its neighboring nodes; therefore, the *mailbox* only needs to save the *e-mails* sent by the neighboring nodes. To this end, this paper presents *neighborhood interaction variable* as *mailbox* to store data information of neighboring nodes.

Definition 9. Neighborhood Interaction Variable. The neighborhood interaction variable refers to a special variable used to asynchronously store the information of the adjacent nodes, referred to as an interaction variable.

Figure 10 shows the mailbox-based neighborhood interaction mechanism based on the network tag. When one node needs to send its data to the neighboring nodes, its *neighbor-tag* first needs to be parsed through the network tag manager, according to the mapping relationship between the *neighbor-tag* and the adjacent node numbers. Then, the transceiver sends the message to the corresponding neighboring node. When the neighboring node sends the data, the data is automatically saved to the interaction variable through the transceiver and the network tag manager; then it waits for the APP to access it based on the *neighbor-tag*.

In summary, the *Neighborhood* programming submodel adopts the parallel programming thinking of *ontology*; that is, when programming the network computing activities, it mainly focuses on two key issues:

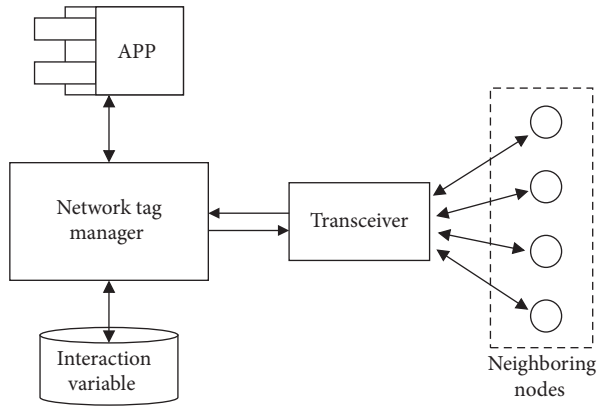


Figure 10: Mailbox-based neighborhood interaction mechanism based on network tag.

(i) Q1: Who am *I*?

(ii) Q2: What should *I* do?

Since decentralized network computing activities can only perform neighborhood interactions, question 2 can be further refined as follows:

(i) Q2.1: Which neighbors should *I* interact with?

(ii) Q2.2: How should *I* handle the data being interacted with?

The *Neighborhood* programming submodel uses the *self-tag* to characterize *I* and the *neighbor-tag* to characterize the neighbors that interact with *I*, and the processing of interaction data (i.e., interaction variables) depends on the actual needs of the network computing activity. In other words, throughout the programming process, the problem is always handled from the perspective of *ontology*. The neighbor is only a nominal interaction object of *ontology*, without specific behavior features and state features, which is also the reason why the programming abstraction *neighbor* only encapsulates the *neighbor-tag*. From the perspective of the overall network, the superposition of the individual operations from every *I* can complete the solution to the

network computing activity. For example, the network summation calculation is a superposition of the operations performed when I is, respectively, inner node, root node, and leaf node.

In the *Neighborhood* programming submodel, the Tag-based programming based on network tags makes the program design of decentralized network computing activities effectively get rid of the dependence on the node number and meets its programming requirements. Specifically, the nodes participating in the network computing activity execute the same program as *ontology* and select the code block in the program to execute according to their corresponding *self-tags*. Interaction with neighboring nodes is guided by the *neighbor-tag*, which means sending the message to the neighboring node that matches the *neighbor-tag* or receiving a message from a neighboring node that matched the *neighbor-tag*. Which nodes are specifically interacting is determined in the APP running phase according to the mapping relationship between the *neighbor-tag* and the adjacent node number, without being specified in the development phase by developers. Therefore, the *Neighborhood* submodel makes the network computing activities get rid of the dependency on the node number or communication address.

4.3. Defining Region in INR. This section first analyzes the control task of network computing activity, then proposes the *Region* programming submodel, and introduces the related concepts, abstractions, and programming mechanisms in detail.

4.3.1. Problem Analysis. The *Neighborhood* programming submodel only describes the interaction behavior of the neighboring nodes from the perspective of *ontology*. Its limited programming vision cannot effectively encapsulate the network computing activities and can only describe the control tasks, including one network computing activity. For more complex control tasks, we need to consider the following issues:

(1) Who is responsible for initiating a network computing activity?

In I^2B control tasks, not all nodes have the right to initiate network computing activities. For example, in the water supply system, for water pressure control task, only the pump *basic unit* is eligible to initiate a network calculation because it directly controls the transmission impetus of the system. Therefore, the initiator must be effectively identified during the programming phase.

(2) Who is eligible to participate in the network computing activities?

For the entire network of I^2B, a control task generally only involves the participation of part nodes; that is, the execution domain of the network computing activity has a boundary, and the boundary is dynamic, which can dynamically change when it exists in a different building. Therefore, the programming model must be able to clarify the execution domain boundaries of network computing activities, that is, determine which nodes are eligible to participate in network computing activities.

(3) How to integrate multiple network computing activities and local tasks in one APP?

A complex I^2B control task may include multiple network computing activities and local tasks of different types of *basic units*, and the execution domain boundaries of different tasks and the requirements for the initiating node may also have large differences. Therefore, the programming model needs to provide an efficient programming mechanism that allows multiple network computing activities and local tasks to be integrated into one APP.

4.3.2. Region Programming Submodel. According to the programming requirements of the I^2B control task, this section proposes the *Region* programming submodel based on the *Individual* programming submodel and the *Neighborhood* programming submodel, as shown in Figure 11.

In the *Region* programming submodel, two programming abstractions of *region* and *basic unit* are introduced. The *basic unit* is an abstraction of the *basic unit* of the building in the physical world, and the *region* is an abstraction of the collection of *basic units* participating in a network computing activity. From the software architecture analysis, the *basic unit* in *Region* submodel is an extension with the network tag on the basis of the *basic unit* programming abstraction in *Individual* submodel and is responsible for not only the processing of local tasks, but also the execution of network computing activities. The originator *basic unit* is a special kind of *basic unit* which is used to initiate the network computing activity. The *region* programming abstraction encapsulates the network computing activities implemented by the *Neighborhood* submodel into its group behavior. *Region* builds group scope and group interaction through constraints and domain variables and the network tag of *basic unit* is set up based on the constraints, which can constrain and manage the *basic units* that participate in and initiate the network computing activities.

(1) Programming Abstractions in Region

Definition 10. Region is an abstraction of the collection of basic units participating in a network computing activity and can be defined as a six-tuple:

$$region = \langle reg_{id}, Ori, NCA_{reg}, C_{reg}, V_{reg}, Str_{reg} \rangle, \quad (7)$$

where reg_{id} represents the name identifier of *region*, specified by the APP developer, unique to one APP. *Ori* indicates the collection of originators. The originator $ori \in Ori$ indicates the *basic unit* that has the right to organize a network computing activity. When there are multiple network computing activities in a *region* and the requirements for the originator nodes are different, each network computing

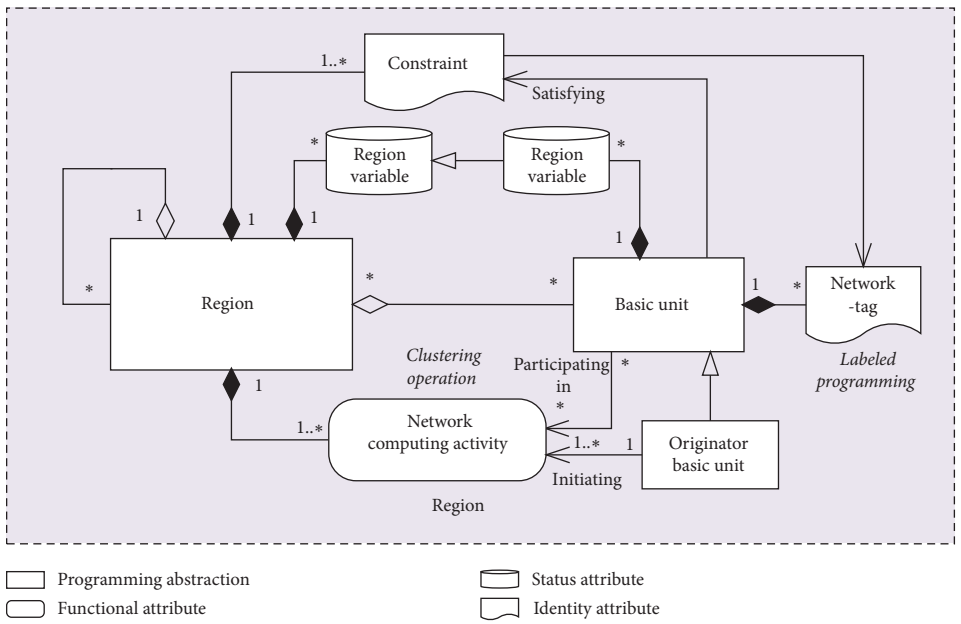


Figure 11: The Neighborhood programming submodel.

activity needs to define a corresponding originator. NCA_{reg} denotes a collection of network computing activity. Network computing activity $nca \in NCA_{reg}$ is a group behavior of *region*. In one *region*, the execution domain boundary of all *nca*s is the same. *nca* can only be initiated by the corresponding originator *ori* in the *region*, and when an *nca* is initiated, all *basic units* in the *region* will automatically participate in the execution of this *nca*. C_{reg} denotes a collection of region constraints. When a *basic unit* meets all C_{reg} of the *region*, it will be automatically added to the *region*. V_{reg} denotes a collection of region variable. Region variable $v_r \in V_{\text{reg}}$ is a variable defined in *region*, which is inherited by all the *basic units* in the *region*. That is, when a region variable v_r_x is defined in the *region*, all *basic units* in the *region* will automatically define a variable whose name is also v_r_x. Therefore, the *region* variable is essentially a vector, i.e., $v_r = \{v_{r1}, v_{r2}, \ldots, v_{rn}\} \in V_{\text{reg}}$, where v_{rn} represents the variable element of the nth node. Region variables are closely related to network computing activities *nca*s, and *nca*s are operations on region variables. For example, a network summation calculation can be considered as summing all elements of a region variable. Str_{reg} denotes a collection of region structure. Region structure $str_{reg} \in Str_{reg}$ is used to characterize the computing network on which the network computing activity is based. $Str_{reg} = \{tree, star, net\}$, where *tree* represents the tree-shaped computing network and there is a strict hierarchical relationship between the *basic units*; *star* represents the star-shaped network (or master-slave computing network), in which, except for the initiator, other *basic units* are equal in status and only interact with the initiator; *net* represents the graphic structure, in which the *basic units* are completely equal in status and free to interact with neighboring *basic units*.

(2) Dynamic Binding and Clustering Operations Mechanism. The relationship between *basic unit* and *region* is dynamic binding. Any *basic unit* that satisfies the constraint rules is automatically added to the *region*. This constraint-based dynamic binding can adapt to changes in the number of participating nodes, making the execution domain boundary of the network computing activity have good dynamics.

The *basic unit* added to the *region* inherits the group attribute (region variable) of the *region* and performs the group duty of *region*. When the *basic unit* with the originating right initiates a network computing activity, all the *basic units* in the *region* will automatically participate in the execution of this network computing activity, which is called the Clustering operations of network computing activities.

There is a many-to-many relationship between *basic unit* and *region*. A *region* can contain multiple *basic units* of different types, and a *basic unit* can also be added to multiple *regions* at the same time. Therefore, multiple network computing activities and local tasks can be effectively integrated by defining several *regions* and different types of *basic units* in one APP. Besides, the *Region* programming submodel also supports nested definitions between *regions*. For example, defining a subregion reg_a in one region reg_b means nesting reg_a within reg_b and is expressed as $reg_a \longrightarrow_{nest} reg_b$. From the set theory perspective, reg_a is a subset of reg_b, so all "*basic units*" bound to reg_a will participate in the group behavior of reg_b, but not vice versa.

The *Region* programming submodel is based on constraints, initiators, and mechanisms of dynamic binding and Clustering operations, which meets the programming requirements for complex control tasks, and has good constructivity and packaging capabilities.

5. Case Study

In this section, we conducted an experiment that applies the language model in an application case to illustrate the developing pattern of the I^2B APP with the programming

model and verify the effectiveness of our approach. The application case is reaching the optimized operation of a variable-air-volume air-conditioning system.

5.1. Case Description. The air-conditioning system is an indispensable part of modern intelligent buildings, and it is also the largest part of energy consumption [33]. The intelligent system achieves the optimal operation of the air-conditioning system through optimization and adjustment and realizes the maximum utilization of energy. Figure 12 shows a simplified structure of the variable-air-volume air-conditioning system to make the introduction of coupling relationship easy to understand; in the system, the variable-air-volume box (VAV BOX) with an adjustable terminal is installed in each room, and the air supply volume of the room can be controlled by adjusting the opening of the VAV BOX valve to meet the needs of each room.

However, since each room shares a ventilation duct, the air supply volume of the room has a strong coupling relationship with other rooms in the pipeline. If the VAV BOX valve opening is adjusted according to the local room demand load, it is easy to cause thermal imbalance [3]; that is, when the total amount of air supplied by the AHU is constant, the valve opening θ_i of any room increases (then the valve impedance S_i is reduced and the air supply volume Q_i is increased), the air supply amount Q_i of other rooms is reduced to a certain extent, and the closer the pipeline distance is, the greater the influence is, i.e., small in the distance and big in the vicinity (in Figure 12 the color depth of the node represents the degree of the impact). Therefore, while adjusting the air supply amount of each room, each room should also notify other rooms to make appropriate adjustments to balance the coupling effect and achieve stable air supply [3].

Actually, the real operation scenario should include more nodes and even an entire floor or building. This section is intended to prove the descriptiveness of the proposed model through describing the case with the models; that is, the model can achieve a generalized and abstract description of a complex control task in I^2B, which can promote the further development and implementation of the APP.

5.2. Programming Requirement Analysis. According to the control requirements, the optimized operation of a variable-air-volume air-conditioning system can be divided into the following steps.

Step 1. Each end room obtains the local demand air volume Q_i.

Step 2. When the demand air volume Q_i of any room changes, adjust the VAV BOX valve opening θ_i of the room and its related rooms.

Step 3. It is time to calculate the total demand air volume Q_{sum} of all end rooms.

Step 4. Adjust the AHU fan speed *Fs* according to the total demand air volume Q_{sum}.

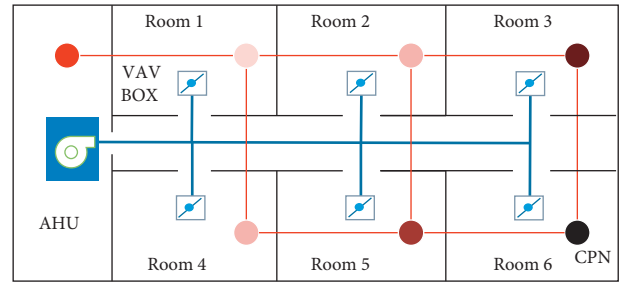


Figure 12: The diagram of the variable-air-volume air-conditioning system.

Obviously, Step 1 and Step 4 belong to the local tasks of the end rooms and AHU, respectively, while Step 2 and Step 3 are two network computing activities, which have significant differences:

(1) The originating node types of the two network computing activities are different. Step 2 is a network computing activity for adjusting the opening of the valve. The type of the initiating node type is a space basic unit, and the initiator is not fixed; that is, the end room where the demand air volume changes has the right to initiate the network computing activity. Step 3 is the network summation calculation. As mentioned above, since only the originating node (the root node) can obtain the final network summation result, the initiator can only be the AHU.

(2) The participating nodes of the two network computing activities have different scopes. The nodes participating in the network summation computing activity in Step 3 include the AHU and all end rooms, and the functional network on which they are based is the functional network of the entire variable-air-volume air-conditioning system. For the network computing activity of Step 2, the participating nodes only include the end rooms. In addition, because the air supply volume influence of each other room has the coupling relationship of being small in the distance and big in the vicinity, the influence sphere of the originating node, i.e., the hop count from the originating node, needs to be restricted. Obviously, there is a significant nesting relationship between the execution domains of the two network computing activities; that is, the execution domain of network computing activity in Step 2 is a subset of the execution domain of the network computing activity in Step 3.

5.3. Developing Pattern Design. Based on the above analysis, Figure 13 shows the developing pattern framework for the optimized operation of variable-air-volume air-conditioning system control task.

The main implementation ideas are as follows:

Firstly, based on the characteristics of the two network computing activities, two *regions* with nested relationships are defined: *Air* and *Air_sub*. *Air* is used to manage the

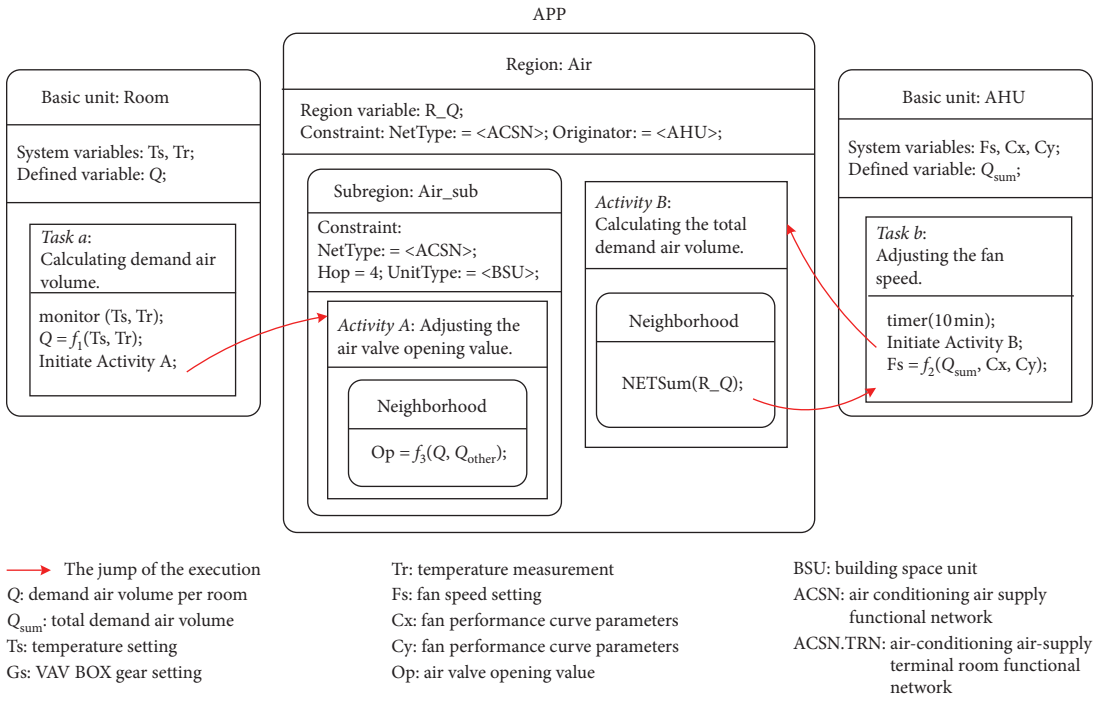


Figure 13: The developing pattern framework for the control task.

network computing activity B for the total demand air volume, whose constraint is *NetType*: = <*ACSN*>, indicating that the functional network is the air-conditioning air supply system functional network; i.e., all *basic units* in this functional network are involved in the activity. The originator of the network computing activity B is AHU; thus, it is necessary to define the initiator in the region *Air*, that is, *Originator ori*: = <*AHU*>. *Air_sub* is a subregion of *Air*, which is used to manage the network computing activity A for adjusting the air valve opening. Its constraints further restrict the basic unit type and hop count based on the region *Air*, i.e., *UnitType*: = <*BSU*>; *Hop*: = 4, which represents a set of building space units within 4 hops from the originator node in the air-conditioning air supply system functional network.

Secondly, two *basic units*, *Room* and *AHU*, are defined. *Room* represents the end rooms of the air-conditioning system inside which a triggered task is defined for real-time monitoring of the room temperature measurement *Tr* and the set value *Ts*. When any one of these two parameters changes, the local demand air volume $Q = f1(Ts, Tr)$ gets a recalculation, and *Room* initiates the network computing activity A for adjusting the air valve opening of the VAV BOX in each room. *AHU* stands for the air handle unit equipment, which is used to realize the timing adjustment of *AHU* fan speed. It internally defines a time-triggered task b, which periodically initiates the network computing activity B for calculating the total demand air volume, and then adjusts the fan speed according to $Fs = f2(Q_{sum}, Cx, Cy)$.

Regions of *Air* and *Air_sub* as well as *basic units* of *Room* and *AHU* are intuitive descriptions of controlled objects in I^{2}B, which means the models of *basic unit* and *Region* are used to describe the typical objects of I^{2}B. In contrast, the *neighborhood* model is not oriented to the domain object and need not be set as a developing element, which substantially is the description of the parallel operation mechanism of I^{2}B. To be specific, the two *neighborhood* models in the developing pattern framework for the case represent two Clustering operations, in which the nodes communicate in a neighborhood-only way.

The APP needs to be downloaded to the CPNs of all *basic units* in the air-conditioning air supply system functional network. The CPN will select and execute the corresponding code module in the APP according to its *basic unit* type. For example, the CPN of the *AHU basic unit* will periodically initiate the network calculation activity B for calculating the total demand air volume through the program entry function (i.e., the automation function) of the *AHU basic unit*, thereby adjusting the local fan speed. The CPNs of the terminal rooms *basic units* will monitor the room temperature measurement and set value through the program entry function of the *Room basic unit* defined in the software. Once the parameters' change occurs, the new local demand air volume is recalculated and then the network computing activity A for adjusting the air valve opening is initiated.

Figure 14 shows the flow diagram of the developing pattern framework, which can explain the executing processes described by the developing pattern framework. The single *basic unit* in I^{2}B is the ultimate controlled object and the *region* can be regarded as a container to simplify Clustering operation with the description support by *neighborhood*. Thus, *basic unit* can be considered as the executing unit. For the *Room basic unit*, it calculates demand air volume locally and participates in a clustering operation for adjusting the air valve opening value. For the *AHU basic unit*, it periodically triggers a clustering operation for

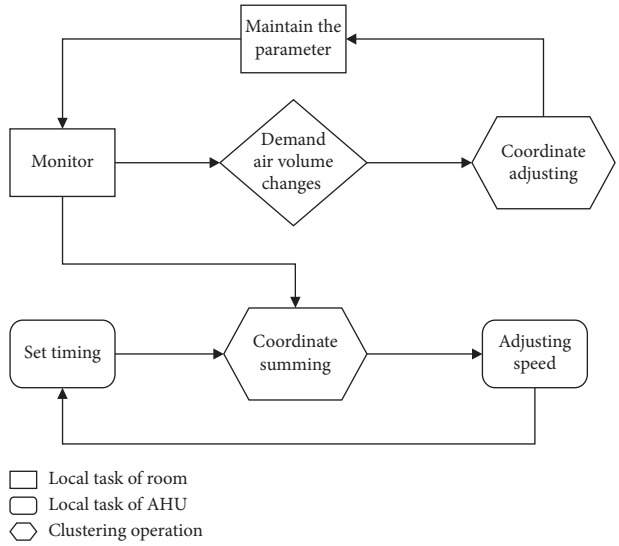


Figure 14: The flow diagram of the developing pattern framework.

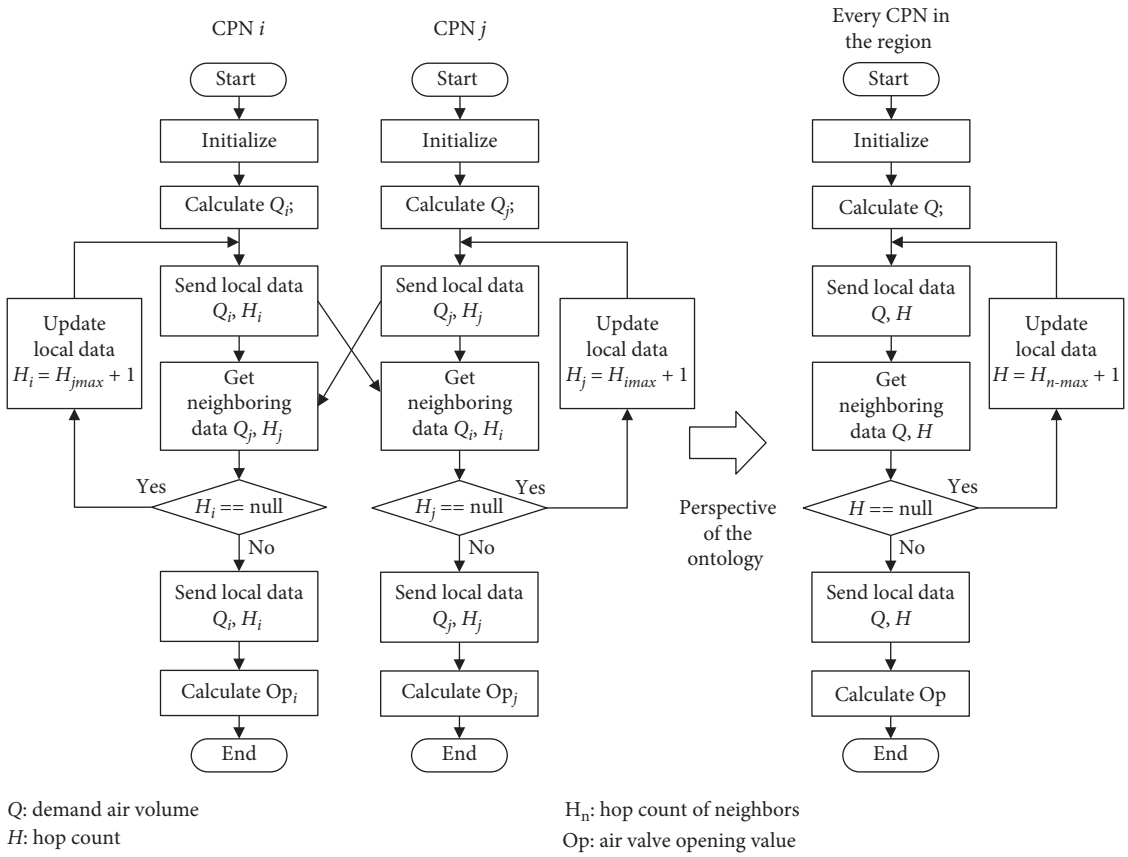


Figure 15: The flow diagram of the clustering operation based on *Neighborhood* submodel.

summing the total demand air volume and then adjusts the fan speed.

It can be seen that the behavior of basic units and regions can be directly described by the submodels of *Individual* and *Region*, while clustering operations are characterized by the *Neighborhood* submodel and imply execution processes. Actually, the interaction between nodes implied from the clustering operations in this case may be the key to the entire case algorithm. Take *clustering adjusting* in Figure 14 as an example; we design the flow diagram of the clustering operation using the *Neighborhood* submodel, which is shown in Figure 15. The interaction between the two neighboring nodes i and j is taken as an example. The key to *clustering adjusting* is making every node know its hop count (H) from

Table 2: The fundamental approaches of the INR programming model.

Requirements	Approaches
Domain characterization	Introducing programming abstractions with I^{2}B domain features such as *slave device*, *basic unit*, *neighbor*, *region*, etc., and effectively integrating functional network and *basic unit* information model into them.
Universality	Programming mechanisms of Tag-based programming and Clustering operations based on slave devices in *Individual* programming submodel.
Parallelism and dynamic interactivity	Programming mechanisms of *ontology* thinking and Tag-based programming based on network tags in *Neighborhood* programming submodel.
Constructivity and encapsulation	Programming mechanisms of dynamic binding and Clustering operations based on network computing activities in *Region* programming submodel.

the changing node, so when the changing node sets its hop count as 0, it is important to spread the number. In the initialization state, every node sets H as null; when it gets a number from its neighbors, it increases the value by 1 then updates its H; when the value of H is not null, the node send its H to neighbors to ensure they can get the data. After that, the node calculates the opening value of air valve and participates in follow-up tasks.

Although i and j are used to distinguish the two nodes here, in actual execution, for every node itself, it only knows itself and its whole neighbor. Therefore, in the self-centered manner, all nodes execute the same procedures and this nondifferent description method can efficiently depict group behavior. The describing idea is described as represented in Section 4.2.2; that is, each node performs the task from the perspective of the ontology, and its neighbors constitute one abstract whole. Through the self-centered interaction of all nodes, the overall goal is gradually realized.

In summary, the proposed model and developing pattern framework for the case can provide a high-level and abstract guide of the variable-air-volume control case. It can be concluded from this application case that the I^{2}B programming model can effectively abstract and describe the I^{2}B control tasks by using the model elements and relationships such as region, basic unit, attributes, and so on. It can encapsulate multiple network computing activities and local tasks into one I^{2}B APP, which can enhance the constructivity and descriptiveness of the applications.

6. Conclusions

In this paper, we propose a programming model for developing I^{2}B APP and motivating the programming language design. Firstly, the INR programming model and its key mechanisms for developing I^{2}B APP are established, which includes three submodels: *Individual*, *Neighborhood*, and *Region*. Then, combining with the programming requirements of the local task, network computing task and I^{2}B control task, the three programming submodels and the mechanisms are introduced, respectively. Table 2 shows the fundamental approaches by which the proposed INR programming model satisfies the requirements of the I^{2}B APP programming.

In addition, we conduct an experiment that applies the INR programming model in an application case of reaching the optimized operation of variable-air-volume air-conditioning system. The case study illustrates the developing pattern of the I^{2}B APP with the proposed programming model and verifies the effectiveness of our approach. The INR programming model and the I^{2}B APP developing pattern based on INR are effective abstractions and essences of the I^{2}B control and development which can further provide meaningful guidance and support for I^{2}B programming language design and I^{2}B APP development.

Data Availability

The data used to support the findings of this study are available from the corresponding author upon request.

Conflicts of Interest

The authors declare that there are no conflicts of interest regarding the publication of this paper.

Acknowledgments

This research was funded by the National Key Research and Development Project of China (New Generation Intelligent Building Platform Techniques) (2017YFC0704100).

References

[1] Q. C. Zhao and Z. Y. Jiang, "Insect intelligent building (I^{2}B): a new architecture of building control systems based on internet of things (IoT)," in *Advancements in Smart City and Intelligent Building*, Q. Fang, Q. Zhu, and F. Qiao, Eds., pp. 457–466, Springer, Singapore, 2018.

[2] Q. Shen, *Studies on Architecture of Decentralized System in Intelligent Building*, Tsinghua University, Beijing, China, 2015.

[3] Y. Dai, Z. Jiang, Q. Shen, P. Chen, S. Wang, and Y. Jiang, "A decentralized algorithm for optimal distribution in HVAC systems," *Building and Environment*, vol. 95, pp. 21–31, 2016.

[4] Z. Jiang and Y. Dai, "A decentralized, flat-structured building automation system," *Energy Procedia*, vol. 122, pp. 68–73, 2017.

[5] A. Hadas and D. Lorenz, "Language oriented modularity: from theory to practice," *The Art, Science, and Engineering of Programming*, vol. 1, no. 2, p. 10, 2017.

[6] T. Kosar, S. Gaberc, J. C. Carver, and M. Mernik, "Program comprehension of domain-specific and general-purpose languages: replication of a family of experiments using integrated development environments," *Empirical Software Engineering*, vol. 23, no. 5, pp. 2734–2763, 2018.

[7] M. Mernik, J. Heering, and A. M. Sloane, "When and how to develop domain-specific languages," *ACM Computing Surveys (CSUR)*, vol. 37, no. 4, pp. 316–344, 2005.

[8] H. Miao, A. Li, L. S. Davis, and A. Deshpande, "Towards unified data and lifecycle management for deep learning," in *Proceedings of the 2017 IEEE 33rd International Conference on Data Engineering (ICDE)*, San Diego, CA, USA, April 2017.

[9] J. Wang, K. He, B. Li, W. Liu, and R. Peng, "Meta-models of domain modeling framework for networked software," in *Proceedings of the International Conference on Grid and Cooperative Computing*, Los Alamitos, CA, USA, August 2007.

[10] A. Kleppe, *Software Language Engineering: Creating Domain-Specific Languages Using Metamodels*, Pearson Schweiz Ag, Zug, Switzerland, 2009.

[11] J. Gray, T. Bapty, S. Neema, D. C. Schmidt, and A. Gokhale, "An approach for supporting aspect-oriented domain modeling," in *Proceedings of the International Conference on Generative Programming & Component Engineering*, Erfurt, Germany, September 2003.

[12] G. T. Xue, Q. L Yang, J. C Xing, and S. Zhao, "Research on programming model for "master-slave distributed tasks" of insect intelligent building," in *Proceedings of the 2018 Chinese Automation Congress*, Xi'an, China, December 2018.

[13] S. Wang, J. Xing, Z. Jiang et al., "A decentralized sensor fault detection and self-repair method for HVAC systems," *Building Services Engineering Research and Technology*, vol. 39, no. 6, pp. 667–678, 2018.

[14] J. Yu, X. Qian, A. Zhao et al., "Decentralized optimization algorithm for parallel pumps in HVAC based on log-linear model," in *Proceedings of the International Conference on Smart City and Intelligent Building*, Springer, Hefei, China, September 2018.

[15] Y. Wang and Q. Zhao, "A distributed algorithm for building space topology matching," in *Proceedings of the International Conference on Smart City and Intelligent Building*, Springer, Hefei, China, September 2018.

[16] Y. Zhang, J. Xing, Q. Zhou et al., "A decentralized state estimation algorithm for building electrical distribution network based on ADMM," in *Proceedings of the 2019 IEEE 15th International Conference on Automation Science and Engineering (CASE)*, IEEE, Vancouver, Canada, pp. 756–761, August 2019.

[17] M. Hossain, A. Islam, and M. Kulkarni, "μSETL: a set based programming abstraction for wireless sensor networks," in *Proceedings of the International Conference on Information Processing in Sensor Networks*, IEEE, Chicago, IL, USA, pp. 354–365, April 2011.

[18] P. Wang, D. Meng, J. Han et al., "Transformer: a new paradigm for building data-parallel programming models," *IEEE Micro*, vol. 30, no. 4, pp. 55–64, 2010.

[19] Y. Zhang, B. Sun, and J. Liu, "A markup language for parallel programming model on multi-core system," in *Proceedings of the International Conference on Scalable Computing & Communications, 8th International Conference on Embedded Computing*, Dalian, China, September 2009.

[20] B. Tekinerdogan and E. Arkin, "ParDSL: a domain-specific language framework for supporting deployment of parallel algorithms," *Software & Systems Modeling*, vol. 18, no. 5, pp. 2907–2935, 2018.

[21] J. Ferber and O. Gutknecht, "A meta-model for the analysis and design of organizations in multi-agent systems," in *Proceedings of the International Conference on Multi Agent Systems (Cat. No. 98EX160)*, IEEE, Paris, France, pp. 128–135, July 1998.

[22] C. Hu, X. Mao, M. Li, and Z. Zhu, "Organization-based agent-oriented programming: model, mechanisms, and language," *Frontiers of Computer Science*, vol. 8, no. 1, pp. 33–51, 2014.

[23] S.-W. Hsiao, C.-H. Lee, M.-H. Yang, and R.-Q. Chen, "User interface based on natural interaction design for seniors," *Computers in Human Behavior*, vol. 75, pp. 147–159, 2017.

[24] P. L. Miseldine, *Language support for process-oriented programming of autonomic software systems*, Ph.D. thesis, Liverpool John Moores University, Liverpool, UK, 2007.

[25] S. Badia, A. F. Martín, and J. Principe, "FEMPAR: an object-oriented parallel finite element framework," *Archives of Computational Methods in Engineering*, vol. 25, no. 2, pp. 195–271, 2018.

[26] R. K. Saha, Y. Lyu, H. Yoshida, and M. R. Prasad, "ELIXIR: effective object-oriented program repair," in *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering*, Champaign, IL, USA, November 2017.

[27] F. Bergenti, E. Iotti, S. Monica, and A. Poggi, "Agent-oriented model-driven development for JADE with the JADEL programming language," *Computer Languages, Systems & Structures*, vol. 50, pp. 142–158, 2017.

[28] A. R. Panisson and R. H. Bordini, "Knowledge representation for argumentation in agent-oriented programming languages," in *Proceedings of the Brazilian Conference on Intelligent Systems (BRACIS)*, Recife, Brazil, October 2016.

[29] A. Metzger, K. Pohl, P. Heymans, and P. Y. Schobbens, "Disambiguating the documentation of variability in software product lines: a separation of concerns, formalization and automated analysis," in *Proceedings of the IEEE International Requirements Engineering Conference*, Delhi, India, October 2007.

[30] D. Dai, Y. Chen, D. Kimpe, and R. Ross, "Trigger-based incremental data processing with unified sync and async model," *IEEE Transactions on Cloud Computing*, vol. 1, 2018.

[31] H. Kasim, V. March, R. Zhang, and S. See, "survey on parallel programming model," in *Proceedings of the IFIP International Conference on Network and Parallel Computing*, pp. 266–275, Shanghai, China, October 2008.

[32] X. J. Mao, *Agent-Oriented Software Engineering: Models, Methodology and Language*, Tsinghua University Press, Beijing, China, 2015.

[33] H. Li, Z. Yu, and W. Liu, "Status of intelligent building development of China—questionnaire analysis," in *Proceedings of the International Conference on Smart City and Intelligent Building*, Springer, Hefei, China, pp. 183–194, September 2018.