Insect Intelligent Building (I2B) is a novel platform of intelligent buildings. The outstanding feature of I2B is the decentralized network structure connected by smart nodes. I2B can employ APPs (applications) developed by various practitioners or programming fans to manage and control buildings. However, due to the unique parallel operation of I2B platform and the popularization of APP developers, there still exists no effective approach to supporting I2B APP development. To deal with the challenges and provide meaningful guidance for describing and developing I2B APP and motivating the prospective programming language design, we propose INR, a programming model for I2B 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 I2B. Finally, we apply the programming model into an application case to illustrate the developing pattern of the I2B APP and verify the effectiveness of our approach.

1. Introduction

The Intelligent Insect Building (I2B) [1–4] is a new type of intelligent buildings. Differing from traditional intelligent building control systems, I2B 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 I2B 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 I2B. In addition to the CPN network hardware system, the I2B 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. I2B 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 I2B control is the APP, but there are some challenges for I2B APP developing.

On the one hand, the APP oriented to decentralized, parallel, and distributed features of I2B is the key to ensure the efficient and stable operation of I2B. However, the new structure and features of the I2B platform also present new difficulties and challenges for I2B application development due to their profound changes. Furthermore, with the expansion of the openness and popularity of I2B, the I2B 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 I2B features and public
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 slave 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 $\mu$SETL, 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:

$$pmo = \langle PA, P_{Mec} \rangle,$$

where $PA$ denotes a collection of programming abstractions. Programming abstractions $PA \subset P_A$ 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 $P_{Mec} \subset P_{Mec}$ are used to describe the interaction between programming abstractions $PA$; that means $P_{Mec} = PA \times PA$.

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 I²B APPs including three submodels, which can be defined as a three-tuple:

\[
INR = \langle \text{Individual}, \text{Neighborhood}, \text{Region} \rangle
\]

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. It can effectively describe the interaction behaviors between a smart node with its neighbor 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 and 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 I²B 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 I²B control tasks and key mechanisms to achieve such domain features as plug-and-play of APPs in I²B. 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 I²B 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.
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_SwitchSet” 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
4.1.2. Individual Programming Submodel. In the I^2B 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^2B 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^2B 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:

\[ unit = \langle \text{name}, \text{type}, D, f_{\text{auto}}, F_{\text{static}}, \text{Dev}_{\text{sla}} \rangle, \]  

where \text{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. \text{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. \( F_{\text{static}} \) represents a collection of basic unit static functions used to characterize the static functional properties of the basic unit. \( \text{Dev}_{\text{sla}} \) denotes a collection of slave devices in basic unit. Slave devices \( d_{\text{sla}} \in \text{Dev}_{\text{sla}} \) are relatively independent members in the basic unit (described in more
detail below). \( f_{\text{auto}} \) 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 \( T = T_{\text{non}} \cup T_{\text{trig}} \), where \( T_{\text{non}} \) represents a collection of nontriggered tasks and \( T_{\text{trig}} \) 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:

\[
d_{\text{sla}} = \langle \text{type}, \text{Tag}, D_{\text{str}}, F_{\text{sla}} \rangle,
\]

where \( \text{type} \) indicates the type of the slave device and is unique to the slave device. \( \text{Tag} \) denotes a collection of slave device tags. \( \text{tag} \in \text{Tag} \), \( \text{tag} \) is used to replace the numbers of slave devices, as defined by a developer during the APP development phase. \( D_{\text{str}} \) indicates the data structure of the slave device, which is used to describe the state attribute of the slave device. The internal data \( d_{\text{str}} \in D_{\text{str}} \) 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 \( d_{\text{Light}} \) 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. \( d_{\text{str}} \) can be accessed in the form of a slave device tag. \( F_{\text{sla}} \) 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 \( f_{\text{auto}} \), 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 \( T_{\text{non}} \) is created in the main execution path of the autonomous function \( f_{\text{auto}} \), it starts execution immediately and will be automatically destroyed after the execution is completed. The next execution still needs an autonomous function \( f_{\text{auto}} \) to make an explicit creation.

After the triggered task \( T_{\text{trig}} \) is created in the main execution path of the autonomous function \( f_{\text{auto}} \), 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 \( T_{\text{trig}} \) 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 \( D_{\text{str}} \) and performance function \( F_{\text{sla}} \) based on slave device tags.

Clustering operations refer to functional operations based on slave device tags and the operations are
automatically applied to all slave devices that own the corresponding tag.

Specifically, when an autonomous function \( f_{\text{auto}} \) calls a slave device performance function \( F_{\text{sla}} \) (or accesses data \( d_{\text{str}} \)) based on a tag (e.g., \( \text{tag}_x \)), all tagged slave devices will execute the performance function (or access data \( d_{\text{str}} \)). 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 I²B 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 I²B, the decentralized network summation calculation is a typical control task based on the needs to be based on the decentralized network of I²B. 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 \( Q_{\text{sum}} \) 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 I²B 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...
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:

$$self = \langle num, V, T_{self}, ON \rangle,$$

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 \(n\)th 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:

$$neighbor = \langle Num, T_{neigh}, R_{n} \rangle,$$

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-tags, used to characterize the neighborhood relationship between self basic unit and neighboring nodes. \(R_{n}\) indicates the mapping relationship between the neighbor-tags and the node numbers.

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

<table>
<thead>
<tr>
<th>Programming requirements for local tasks</th>
<th>Approach</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dependency with basic unit types</td>
<td>Type attribute of the basic unit</td>
</tr>
<tr>
<td>Flexibility of the slave device control and independence of the device number</td>
<td>Tag-based programming and clustering operations of the slave devices</td>
</tr>
<tr>
<td>Independence and autonomy</td>
<td>Autonomous function of the basic unit</td>
</tr>
<tr>
<td>Parallel and triggered execution</td>
<td>Task-based parallel programming and triggered mechanism</td>
</tr>
</tbody>
</table>

![Figure 7: The pseudocode of implementing the network summation based on the traditional parallel programming model.](image-url)
of adjacent basic units, i.e., \( R_n \subseteq \text{Num} \times T_{\text{neigh}} \), 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 I²B network task.

**Definition 8. Neighbor-Tag.** The neighbor-tag refers to the collection of neighborhood relationships between the node participating in the I²B 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 (NS5) then send \((Q, N3)\) in Figure 7 can be replaced with if (self-tag is "leaf") then send \((Q, \text{"father"})\). 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 I²B 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 I²B, there may be hundreds of nodes participating in a network task; thus, this mutual...
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 I2B.

In I2B, 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:

(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

---

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

**Figure 10:** Mailbox-based neighborhood interaction mechanism based on network tag.
network computing activity. For example, the network summation calculation is a superposition of the operations performed when \( l \) 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 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 \( \text{I}^2\text{B} \) 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 \( \text{basic unit} \) 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 \( \text{I}^2\text{B} \), 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 \( \text{I}^2\text{B} \) control task may include multiple network computing activities and local tasks of different types of \( \text{basic units} \), 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 \( \text{I}^2\text{B} \) 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:

\[
\text{region} = \langle \text{reg}_{\text{id}}, \text{Ori}, \text{NCA}_{\text{reg}}, \text{C}_{\text{reg}}, \text{V}_{\text{reg}}, \text{Str}_{\text{reg}} \rangle, \quad (7)
\]

where \( \text{reg}_{\text{id}} \) represents the name identifier of region, specified by the APP developer, unique to one APP. Ori indicates the collection of originators. The originator \( \text{ori} \in \text{Ori} \) 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...
activity needs to define a corresponding originator. \(NCA_{\text{reg}}\) denotes a collection of network computing activity. Network computing activity \(nca \in NCA_{\text{reg}}\) is a group behavior of region. In one region, the execution domain boundary of all \(nca\) 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 \(nca\). \(C_{\text{reg}}\) denotes a collection of region constraints. When a basic unit meets all \(C_{\text{reg}}\) of the region, it will be automatically added to the region. \(V_{\text{reg}}\) 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., \(v_{r} = \{v_{r1}, v_{r2}, \ldots, v_{rm}\} \in V_{\text{reg}}\), where \(v_{rn}\) represents the variable element of the nth node. Region variables are closely related to network computing activities \(ncas\), and \(ncas\) are operations on region variables. For example, a network summation calculation can be considered as summing all elements of a region variable. \(Str_{\text{reg}}\) denotes a collection of region structure. Region structure \(str_{\text{reg}} \in Str_{\text{reg}}\) is used to characterize the computing network on which the network computing activity is based. \(Str_{\text{reg}} = \{\text{tree}, \text{star}, \text{net}\}\), where \(\text{tree}\) represents the tree-shaped computing network and there is a strict hierarchical relationship between the basic units; \(\text{star}\) 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; \(\text{net}\) 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 \(reg_{b}\) in one region \(reg_{a}\) means nesting \(reg_{a}\) within \(reg_{b}\) and is expressed as \(reg_{a} \rightarrow_{\text{nest}} reg_{b}\). 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 I²B 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 $\theta_i$ 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 $Q_j$ 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 I2B, 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 $\theta_i$ 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 $F_5$ according to the total demand air volume $Q_{sum}$.

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
network computing activity A 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 A 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 $Q = f_1(Ts, Tr)$ 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 $Fs = f_2(Q_{sum}, Cx, Cy)$.

Regions of Air and Air_sub as well as basic units of Room and AHU are intuitive descriptions of controlled objects in $I^2B$, which means the models of basic unit and Region are used to describe the typical objects of $I^2B$. 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 $I^2B$. 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 $I^2B$ 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

<table>
<thead>
<tr>
<th>Basic unit: Room</th>
<th>Basic unit: AHU</th>
</tr>
</thead>
<tbody>
<tr>
<td>System variables: $Ts$, $Tr$; Defined variable: $Q$;</td>
<td>System variables: $Fs$, $Cx$, $Cy$; Defined variable: $Q_{sum}$;</td>
</tr>
<tr>
<td>Task-a: Calculating demand air volume; monitor ($Ts$, $Tr$); $Q = f_1(Ts, Tr)$; Initiate Activity A;</td>
<td>Task-b: Adjusting the fan speed; timer(10min); Initiate Activity B; $Fs = f_2(Q_{sum}, Cx, Cy)$;</td>
</tr>
<tr>
<td>Subregion: Air_sub</td>
<td>Neighborhood</td>
</tr>
<tr>
<td>Constraint: NetType: = &lt;ACSN&gt;; Hop: = 4; UnitType: = &lt;BSU&gt;;</td>
<td>Activity B: Calculating the total demand air volume.</td>
</tr>
<tr>
<td>Defined variable: $Q_{sum}$;</td>
<td>Neighborhood</td>
</tr>
<tr>
<td>Activity A: Adjusting the air valve opening value.</td>
<td>NETSum($R$, $Q$);</td>
</tr>
<tr>
<td>Neighborhood</td>
<td></td>
</tr>
<tr>
<td>Op = $f_3(Q, Q_{other})$;</td>
<td>BSU: building space unit</td>
</tr>
<tr>
<td>Tr: temperature measurement</td>
<td>ACSN: air conditioning air supply</td>
</tr>
<tr>
<td>Fs: fan speed setting</td>
<td>functional network</td>
</tr>
<tr>
<td>Cx: fan performance curve parameters</td>
<td>ACSN.TRN: air-conditioning air-supply terminal room functional network</td>
</tr>
<tr>
<td>Cy: fan performance curve parameters</td>
<td>Op: air valve opening value</td>
</tr>
</tbody>
</table>

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

![Flow Diagram](image-url)
In this paper, we propose a programming model for developing I2B APP and motivating the programming language design. Firstly, the INR programming model and its key mechanisms for developing I2B 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 I2B 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 I2B 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 I2B APP with the proposed programming model and verifies the effectiveness of our approach. The INR programming model and the I2B APP developing pattern based on INR are effective abstractions and essences of the I2B control and development which can further provide meaningful guidance and support for I2B programming language design and I2B 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


<table>
<thead>
<tr>
<th>Requirements</th>
<th>Approaches</th>
</tr>
</thead>
<tbody>
<tr>
<td>Domain characterization</td>
<td>Introducing programming abstractions with I2B domain features such as slave device, basic unit, neighbor, region, etc., and effectively integrating functional network and basic unit information model into them.</td>
</tr>
<tr>
<td>Universality</td>
<td>Programming mechanisms of Tag-based programming and Clustering operations based on slave devices in Individual programming submodel.</td>
</tr>
<tr>
<td>Parallelism and dynamic</td>
<td>Programming mechanisms of ontology thinking and Tag-based programming based on network tags in Neighborhood programming submodel.</td>
</tr>
<tr>
<td>Constructivity and encapsulation</td>
<td>Programming mechanisms of dynamic binding and Clustering operations based on network computing activities in Region programming submodel.</td>
</tr>
</tbody>
</table>


