Hindawi
Scientific Programming
Volume 2020, Article ID 9634389, 18 pages
https://doi.org/10.1155/2020/9634389

Research Article

A Graphical Programming Language and Its Supporting Tool for Insect Intelligent Building

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

[1]*College of Defense Engineering, Army Engineering University of PLA, Nanjing 210007, China*
[2]*China Xi'an Satellite Control Center, Xi'an 710000, China*

Correspondence should be addressed to Shuo Zhao; zephyrsure@outlook.com

Received 24 February 2020; Revised 11 September 2020; Accepted 14 October 2020; Published 16 November 2020

Academic Editor: Danilo Pianini

The emerging Insect Intelligent Building (I^2B) platform is pioneering a new realm in intelligent buildings. I^2B has a distributed and decentralized network structure with intelligent nodes, and the key enabler is an application (APP) that functions to process information from intelligent nodes and accomplish complex control tasks in a decentralized network. To develop APPs for I^2B, a proper programming language is the foremost goal; however, existing programming languages cannot be applied directly due to I^2B's unique structure and distinction of application domains. This paper aims to provide language support for a direct and friendly development of I^2B APPs. We propose a graphical programming language that adapts to the operating characteristics of I^2B and users' habits of buildings. Specifically, we first analyze the domain characteristics of the proposed language and present a general programming language model illustrated with a motivating example. Then, we investigate the detailed design, definition, and usage of graphic elements. Additionally, we implement a prototype support tool for the graphical programming language to develop I^2B APPs. Moreover, we use the application example of a building's constant pressure control for evaluating the effectiveness of our work.

1. Introduction

Intelligent buildings can provide users with an efficient, comfortable, and convenient user-centric building environment by integrating modern science and technology and optimizing the structure and management of buildings [1]. Along with the continuous improvement of people's comfort and safety requirements and the increasing control demands of buildings, traditional intelligent building control systems are no longer sufficient. A decentralized building platform named Insect Intelligent Building (I^2B) has been developed as a new realization form of an intelligent building system, with the advantages of self-organization, plug-and-play, distribution, and so on [2]. The application (APP) of I^2B is the control program to control and manage buildings and provide building services. Therefore, the development of the APP is a key part to ensure the efficient and stable operations of I^2B. This also means that a friendly programming language is required for the APP developers in the I^2B community. Furthermore, in the development process, this kind of programming language can adapt to the domain needs of the APP developers and easily build a bridge of communication between the system and users. Consequently, investigating a programming language that can better meet the requirements of APP developers and I^2B running features is an important problem.

The mainstream programming languages are mainly divided into four parts: assembly language, C and other high-level native languages, and virtual machine languages [3–5]. However, existing programming languages are not suitable for the development of I^2B. On one hand, the architectural engineers and building owners are the major users in the stages of building configuration and usage. They need the programming language to develop an equipment control strategy and realize personalized customization, but they do not have strong programming ability. Assembly language, high-level native languages, and virtual machine languages are mainly oriented to the computer level, concentrate on performance objectives, and therefore seek efficiency at the expense of humanization and aesthetics. Moreover, professional usage specifications with high learning and implementation costs result in a heavy burden on building users.

On the other hand, domain objects of buildings are the key elements that programming and development efforts should focus on. Moreover, the distributed, parallel, and plug-and-play characteristics of the I^2B place high requirements on the development of the APP. Therefore, while traditional programming languages are comprehensive, they are not able to explain the building control in a proper way.

Domain-specific languages (DSLs), which are oriented to a specific domain and can clearly describe the domain objects, have been applied in certain fields [6, 7]. Among many types of programming languages, graphical programming languages are characterized by strong readability and simple language rules, which can greatly reduce the threshold of programming. After long-term development, graphical programming languages have been widely accepted and applied in various application scenarios, forming many graphical domain-specific programming languages [8–10]. Although the existing Scratch [11] for education and G [12] for measurement and control are very popular, they are difficult to meet the needs of I^2B application development in terms of comprehensibility and readability of graphic elements, hardware performance requirements, parallelism, etc. In addition, these languages have limitations in describing the control behaviors of I^2B due to the different domain requirements and intentions.

The existing programming languages cannot be efficiently applied to developing APPs for I^2B. Therefore, establishing a new programming language that adapts to the I^2B platform and developer habits of buildings becomes an important and challenging issue.

To deal with these challenges, the existing work [13] proposed a graphical programming language for the decentralized building intelligent system and described the design of graphic elements and their interfaces. Although this language can describe a case of constant pressure control of water supply system with the designed graphic elements, the design process lacks credible support and guidance, and the proposed language lacks support and verification of actual tools, which makes the language difficult to become an effective development system.

In this paper, we propose a new optimized graphical domain-specific programming language for I^2B. We conduct an analysis from the aspects of domain characteristics and the language model to support the establishment of language elements; moreover, we design and develop a support tool for the I^2B graphical programming language and make some verification. The main contributions are as follows:

(1) To the best of our knowledge, we are the pioneer that establish the programming models for I^2B based on its characteristics and programming requirements

(2) We design a kind of graphical domain-specific programming language for I^2B and provide the definition and usage of graphic elements

(3) We implement a prototype support tool for the language that can support graphical programming and graph-to-text conversion

(4) We conduct an experiment that applies the proposed language in an example application of reaching constant pressure control of a building's water supply system to verify the effectiveness and a comparative experiment to evaluate the efficiency of our work

The rest of paper is organized as follows. In Section 2, we introduce the background of I^2B and related work. We also provide a motivating example to lead the explanation of this paper. Then, we analyze the domain characteristics and programming requirements of I^2B in Section 3. We present the domain model and programming model as general programming language models of I^2B in Section 4. In Section 5, we discuss the domain-specific graphical programming language for I^2B from the aspects of the design concept, graphical definition, and formal definition of language elements. Then, we provide the implementation process of the prototype support tool for the language in Section 6, and an example application using the language and its support tool with some discussion is presented in Section 7. Finally, we conclude the paper in Section 8.

2. Background and Related Work

2.1. Insect Intelligent Building and Its Application Development Method. The Insect Intelligent Building [2, 14, 15] is a new type of intelligent building with a decentralized network structure. I^2B is based on the spatial distribution of buildings and regards the whole building with its electromechanical system as a decentralized network connected by building space units and large electromechanical equipment. Figure 1 shows the structure of I^2B. Each building space unit and electromechanical equipment corresponds to an intelligent node called CPN (computing process node), which contains a standard information model and can be standardized and produced in large quantities. The standard information model integrates and manages various types of information of the building space unit and large electromechanical equipment.

CPNs are connected in a plug-and-play manner according to the topological relationship of building space, and all CPNs are equal to each other, forming a decentralized network. The interconnected CPNs conduct parallel computing based on network communication, constituting a decentralized parallel computing platform.

As shown in Figure 1, the APP store for I^2B is an algorithm library and information platform in the cloud containing a mass of APPs, which encapsulate intelligent building control algorithms. The APP store is also an open community in the intelligent building domain, in which developers can upload their APPs while the operation and

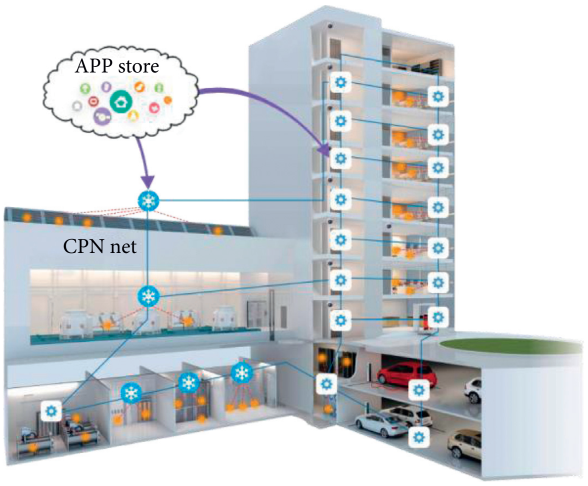


Figure 1: The structure of I^{2}B.

maintenance manager or building users can download suitable APPs from the store to make the building run more intelligently.

At present, the I^{2}B project has achieved some remarkable results, such as the intelligent control node hardware CPN, the operation system of I^{2}B TOS (Things' Operating System), and the development and implementation of various I^{2}B control algorithms. Moreover, some applications and tests of practical projects have been realized and the advantages of the I^{2}B platform have been verified.

I^{2}B solves the problems of "practicality" and "universality" of the traditional centralized building automation system in practical applications. Its decentralized structure transcends the traditional system design model, eliminating the need to establish a central monitoring station, enhancing the control system's fault tolerance and ability to cope with changes. I^{2}B has the characteristics of efficient sharing, self-identification, self-organization, self-coordination, easy operation, easy transformation, and easy expansion, which effectively solves the difficulties faced by traditional centralized building control systems and meets the new needs of modern intelligent buildings.

TOS is a distributed operating system dedicated to the I^{2}B platform, which is used for the scheduling of computing resources, task process management, and the operation of I^{2}B algorithm. TOS provides a series of APIs (Application Programming Interfaces). Through these APIs, functions such as variable management, definition or triggering of the sequence of computation tasks, and obtaining calculation results can be realized [16, 17]. By using API instruction sequences to define parallel computing task sequences, the control and management ideas of construction engineers can be transformed into the code that the CPN computing network can understand and execute.

The I^{2}B application is a program that realizes real-time optimization control and data analysis of building space and electromechanical equipment. It is a parallel computing task sequence based on the editing API provided by TOS. The API interface provided by TOS is open, so developers can use C language to edit the applications on such an operating system according to the building control logic algorithm, then compile the C program through TOS, and download it to the CPN platform to run to realize building control and management. Various control management functions in the building, such as air conditioning system optimization control [15], chiller group control [18], and sensor fault diagnosis [19], can be developed as applications downloaded and operated in TOS.

However, the programming method based on C language pays too much attention to the low-level details and code structure rather than the top-level application logic, and the programming abstraction level is low. Developers need to build I^{2}B applications from the bottom and rely on experience to ensure the rationality of the program structure. The programming work is cumbersome, and the program is not easy to modify and maintain. Furthermore, the C language does not provide language elements corresponding to the I^{2}B domain abstractions, so modeling I^{2}B domain objects will be very complicated. Therefore, the application development method based on C language requires developers to have high programming skills and I^{2}B domain knowledge, which makes the threshold for developing applications higher, and it is difficult for basic users without development experience to join in, resulting in overall development inefficiencies and waste of resources.

With the expansion of the openness and popularity of I^{2}B, the personnel engaged in I^{2}B development activities are no longer limited to system engineers, but gradually expand to the public level of operation and maintenance managers, households, etc., and the most easily accepted and understood by the public is the way of graphic programming. Therefore, how to pay full attention to application developers and provide them with a friendly, simple, intuitive, low-cost, and efficient graphical programming language has become a new challenge and requirement for I^{2}B.

2.2. Graphical Programming Language and Building Control. To introduce related work, we first recommend several popular graphical programming languages and tools. Then, we delve into the programming languages in the field of building control. Furthermore, we introduce some academic studies on architectural design stage and intelligent living environment.

In the field of graphical programming language and applications, there are some well-known research works on products: Scratch [11], a graphical programming language and tool developed by Massachusetts Institute of Technology, is designed for children to program in a simple and easy way. The main function of Scratch is to achieve small animations or small games by programming and operating pictures on the computer. Google Blockly [20] is a graphical programming language developed by Google for children's programming education. Built on Web technology, it edits applications by dragging and dropping programming blocks, similar in function and design to Scratch. Although Scratch and Google Blockly are widely used in education, the shape of their programming blocks does not contain semantics, and children need to understand the words on the programming blocks to distinguish their functions, so the intelligibility of programming blocks is not high. In addition, because Scratch uses a control flow language with low parallelism, it is difficult to support the parallel computing mode required by the I^2B CPN network. Squeak Etoys [21] is a programming teaching environment with a simple and powerful ability to describe object models. The program is written by splicing graphic blocks, but the graphic blocks are also designed in the form of text, which makes the program not intuitive and understandable. Language G [12], a graphical programming language of LabVIEW, developed by National Instruments, uses virtual instrument (VI) components to program the control of instruments and data acquisition. Because the basic elements of Language G used in programming are terms and icons familiar to test engineers, such as various knobs, switches, and waveforms, the interface is very intuitive. However, LabVIEW has very high requirements on the hardware performance of access devices. For example, sensor data input needs to be completed through a special data acquisition card, so it is difficult for ordinary low-power sensor devices and CPNs of I^2B to access LabVIEW. NXT-G [22] is a graphical programming software developed by Lego. It uses a data flow language and can simply connect physical devices to build applications. However, NXT-G is a dedicated language and only focuses on motor control, so it is difficult to support CPN network programming and cannot be expanded by itself, so its actual ability is very limited.

Although these languages are easy to use and popular, in general, the domain they focus on is not intelligent buildings, and they cannot effectively describe building control tasks. Therefore, applying these languages to I^2B would give rise to a mismatch of the fields, more cost, and lower efficiency.

In the field of building control, most automatic control applications of traditional buildings are mainly implemented by PLC (programmable logic controllers). There are three kinds of PLC graphical programming languages, including the ladder diagram (LD) based on electromechanical components, the function block diagram (FBD) based on electronic components, and the sequential function chart (SFC) based on low-level programming. They are defined in the IEC61131-3 standard [23] and have been widely accepted and applied in the control community. With the increasing requirements of decentralized and intelligent system control functions, the IEC 61499 standard has emerged. The new standard defines the service-interface function block (SIFB), which effectively expands the communication capabilities of IEC 61131-3 in distributed systems, enabling distributed PLC control to have a truly implemented path [24, 25]. However, today's smart buildings, especially I^2B, from design and construction to maintenance and management, involve more devices and more complicated control processes with equipment, the environment, and the parallel network. The graphical programming languages for PLC are limited by the fixed hardware and mode of the PLC and neither can be flexibly applied to other control objects, nor can they support the programming development of abstract objects for building space units and electromechanical devices in I^2B. Moreover, they are mainly for electrical engineers, and the threshold is high, making it difficult to provide good friendliness and ease of use.

At the stage of architectural design, Khan et al. [26] propose a set of rules-based algorithms that used visual programming language (VPL) to import elements such as adaptive components, beams, attics, and home components as CSV files to automatically generate geometric conditions in BIM and visualize the potential risks and safety resources installation. However, its graphic elements mainly represent some entities in civil engineering construction, which are different from physical objects such as space units and air conditioning systems in building control to some extent, and VPL is difficult to represent distributed and parallel computing characteristics. Kang et al. [27] propose an effective BIM-integrated object query method using LandXML and IFC and define object types and a simple query language required for object queries. Although the basic elements of the query language include building members, services facilities such as air conditioning systems and object types, and attributes such as total area related to building space planning, it uses a syntax definition similar to the SQL SELECT clause, and the characteristics of graphical programming are not strong.

In addition, some researchers focus on the living environment in an intelligent building. Desolda et al. [28] propose new operators for defining rules combining multiple events and conditions and constraints on rule activation for building smart objects. Although it can provide end users with interactive tools that allow them to customize smart spaces without being forced to understand technical issues, the differences between smart objects and CPNs and the heterogeneity of data formats make it difficult to apply to I^2B application development. Caivano et al. [29] propose We@Home, a mobile application for smart home control, with gamification elements to manage the smart objects in their smart home. Its main feature is to enable a group of people living in the same environment (such as a family) to collaborate, while in I^2B the user usually controls the device individually, so this approach is not suitable for I^2B. Ardito et al. [30] present the task automation tool EFESTO based on the Internet of Things technology and the End User Development (EUD) paradigm, which offer novel visual interaction paradigms to enable end users to easily express rules for smart object configuration. However, this tool is mainly aimed at elderly users and the configuration rules of smart objects are very simple, which is of limited reference value for users who need to develop relatively complex I^2B applications.

These studies significantly improve the efficiency and quality of the architectural design and construction management phase. However, due to their different characteristics and different orientations to domain requirements, these research results cannot be directly applied to the field of building control. In particular, the characteristics of I^2B and its users enact more specialized requirements than those in the above studies.

Accordingly, the existing programming languages cannot provide direct development methods and suitable control task descriptions for I^{2}B, making it difficult to achieve an efficient development and programming of I^{2}B. Therefore, an efficient, practical, and friendly programming language support for I^{2}B is needed.

2.3. Motivating Example. APPs run on the platform of I^{2}B to implement the device control and optimizing operation algorithms. We provide a motivating example to lead the explanation of this paper. The example case involves the constant pressure control of a building's water supply system [14].

(1) Case description: the water supply system is one of the most basic systems in buildings. To ensure the quality of the water supply for large buildings and avoid the problem of insufficient water pressure caused by large-scale horizontal and vertical pipe networks, the water supply system is equipped with a pressure pump. The pump controls its rotating speed according to the return value of the water pipe pressure at each place and completes the increase or decrease of the pressure to ensure the water supply is stable, safe, and reliable.

The typical water supply system in buildings is shown in Figure 2. Water pipes, pipe ends, and the pump make up the water supply functional subnet. The ends of the water pipe network are distributed within the building space units and respond to different CPNs, while the pump responds to another a certain CPN. The pressure sensor belongs to a CPN of the space unit, and its measurement parameter is bound to the standard information model. Although it is not possible to measure the pressure values at every position of the water pipe, the measurement values of each end can be obtained through the standard information model of the space unit. The water pump can obtain the values transmitted through the network to monitor and adjust the pressure of the water pipe network.

(2) Control strategy: regularly initiating pressure testing is performed. If the water pressure at the end is lower than the set value and continues for a period of time, the pump rotation speed should be increased. When the amount of the low-pressure ends increases, the rotation speed should be increased more.

Analogously, the pressure reduction control is performed with the opposite process.

The pump rotation speed is proportional to the sum of the deviation between the water pressure and the set value.

(3) Developing difficulties: the water supply system is based on the platform of I^{2}B. To adapt to the specific system structure, some difficulties are encountered to develop the program:

D1: how to achieve control of the pump and the space unit?
D2: how to achieve acquisition of the end pressure?

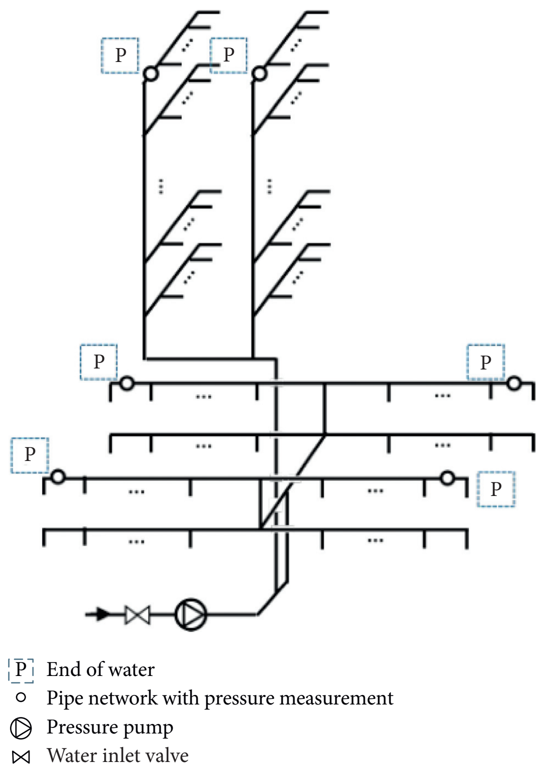


Figure 2: The typical water supply system in buildings.

D3: how to achieve integrated operation of the water supply system?
D4: how to achieve operation and calculation of information?

3. Domain Analysis for Programming in I^{2}B

3.1. Domain Characteristics Analysis for APP Programming. In response to the above problems, this paper presents our approach to design and develop a graphical programming language and tool for I^{2}B. To build a DSL, it is necessary to understand the characteristics of the domain. The School of Architecture, Tsinghua University [31–33], and China Academy of Building Research [34, 35] conducted a series of on-site investigations on the current state of the commercial public intelligent buildings in China. These investigations point out the existing problems and analyze the reasons. The human factor is a very important reason, where the most obvious part is the construction personnel and management personnel. In addition, because the direct users of the programming language are the people in the buildings, they are the source as well as the purpose of the language design. Therefore, this section consults the relevant investigation results of the intelligent buildings and combines the findings with the research work of I^{2}B. We first analyze the users' characteristics and requirements, and then summarize the programming requirements for the new characteristics and request of the I^{2}B platform. As shown in Table 1, the analysis objects of the domain are the users of I^{2}B and the compositions of I^{2}B itself. The programming requirements for the users of I^{2}B are as follows:

Table 1: Domain analysis of I^{2}B.

Domain object		Characteristic analysis
Users	Architectural engineer	Weak programming ability, complex requirements of construction task, and knowing the control of buildings well
	Building management personnel	Short of programming ability, comprehensive management, fast and convenient work, familiar with the situation, and usage of building services
	Room owner	Short of programming ability and personalized customization needs of space environment
	Program developer	Lack of understanding of buildings and I^{2}B and strong programming skills
I^{2}B	Decentralized network structure	Parallel tasks, distributed computing, and neighbor interaction
	Building environment	Environment sensors and event-driven
	Information model of buildings	Bound to basic unit and direct access
	APP store	Separation of “hard and soft,” generalization, and reusability

(1) Architectural engineers: they complete the key work in the configuration stage of the building construction, such as HVAC, water supply, and distribution. They know everything about the building control systems but have no such strong programming ability. They need to implement complex building control tasks with the requirement of efficient, easy, and convenient development.

(2) Building management personnel: managing an intelligent building cannot be separated from building management personnel. They are the people who perform the work of operation, maintenance, and management to maintain the normal operation of the building. They do not possess computer abilities and know little about programming, whereas they have familiarity with the situation and usage of building services. Comprehensive management and urgent restoration are their usual jobs; thus, an easy, fast, and convenient operation method is their requirement.

(3) Room owners: room owners may be the inhabitants in a residential building and the companies or people in a commercial building. Most owners do not have programming abilities but have room configuration requirements that entail personalized customization needs of the space environment.

(4) Program developers: I^{2}B is an open platform that welcomes everyone to become involved. Program developers have strong programming skills and rich development experience and provide a source of vitality for development, although they do not belong to the domain of I^{2}B. The lack of knowledge of buildings and I^{2}B may become a barrier, so they need a development method with a clear and understandable description of buildings.

Although the users have different identities, the majority, as end users, do not have strong programming ability but have a fair understanding of buildings. Hence, an easy, abstract, and friendly language is requisite; meanwhile, the language should be able to intuitively depict buildings for program developers who know little about I^{2}B.

The programming requirements for the platform of I^{2}B are as follows:

(1) Centerless network structure: the structure of I^{2}B is a decentralized network and every CPN is equal without a control center. Moreover, every CPN only connects and interacts with its neighbors to achieve distributed computing. Based on this structure, the control strategy should also be designed in a distributed and parallel way.

(2) Building environment: the environment is an important part of the building, reflects users' requirements, and carries the system's action. Changes in the environment lead to changes in the control strategy, which is regarded as an event-driven task [36]. There are a mass of sensors in a building to measure the environmental condition, which can be the driving sources of the event-driven task.

(3) Information model of buildings: the standard information model is bound to every basic unit, representing all parameters of terminal devices in the space unit and electromechanical equipment as well as writing data to terminal devices. Through the standard information model, the system can directly access the data of devices to control them.

(4) APP store: the APP store is the center where various kinds of control algorithms are stored and is separate from the structure of I^{2}B. The separation of hardware and software contributes many benefits, making APP development independent from concrete hardware platforms [37]. Moreover, it raises the claim that the APP should be general and reusable.

The compositions of I^{2}B indicate that some basic operating mechanisms, such as network parallelism, event triggers, the open interface, and software, are unbundled. Moreover, the basic mechanisms of I^{2}B should be embodied by running a language that has parallel-supported, event-driven, open and scalable, and platform-independent characteristics.

3.2. Analysis of Programming Requirement. According to the domain characteristic analysis of I^{2}B, a conclusion can be drawn that the graphical programming language for I^{2}B should follow two basic rules defined as follows.

End-user oriented: fully considering the user's background factors [38], focusing on key people with weak programming skills, making the language highly abstract, simple, reasonable, easy to read, and easy to use, providing a totally friendly way for visualization, thinking, and operation.

Domain-specific: designing from the internal and external characteristics of the domain, including the following two points:

(1) Construction features: combining the external performance and domain-specific elements, such as the room, AHU (air handle unit), water pump, and distribution box, and describing buildings in a clear, friendly, and comprehensive manner.

(2) I^{2}B operating features: combining the intrinsic performance and addressing the operating principles with parallelism, distribution, and centralism.

4. General Programming Language Model

The core elements of a graphical programming language are the static graphic element and dynamic connection relations. Combined with the domain analysis of I^{2}B, in this section, we first conclude the domain model to state the key elements of I^{2}B and the programming language. Then, based on the domain model, we describe the relationship of programming language domain elements by building the programming model.

4.1. Domain Model. The domain model is a visual representation of the concept classes and objects in the Insect Intelligent Building. By abstracting the specific elements in the Insect Intelligent Building and representing the relationship between domain elements, the association between the programming language elements and the elements of I^{2}B is facilitated. The domain model is shown in Figure 3.

Definition 1. Basic Unit is the fundamental element in I^{2}B, representing two kinds of CPNs that correspond to the building space unit and electromechanical equipment.

Definition 2. Terminal represents the *Terminal* devices in *Basic Units* of building space units and large electromechanical equipment and is bound to the standard information model.

Each *Basic Unit* can have multiple *Terminal* elements, as the building space unit has *Terminal* devices and the electromechanical equipment has controlled components. Referring to the motivating example of the water supply system in Figure 2, the pump and every space unit can be called *Basic Units*. Furthermore, the end of pipe is within the space unit; thus, it can be regarded as a *Terminal* of the *Basic Unit*.

Each *Basic Unit* has a standard information model that contains multiple information model elements for storing parameters and status data. The data depends on the *Terminals* in the *Basic Unit*, and each *Terminal* may provide several parameters, such as the pressure of the end pipe is a parameter of the standard information model in the building space *Basic Unit*.

Definition 3. Region is a relatively independent and complete functional network composed of multiple *Basic Units* in the intelligent building.

Definition 4. Terminal Group is a collection of some of the same types of *Terminal* devices.

Some *Terminal* elements that belong to the same *Basic Unit* element can form a *Terminal Group* for simultaneous control, and all the *Terminals* are inherited from the *Terminal Group*. Analogously, some associated *Basic Units* can form a *Region*, and every *Basic Unit* is inherited from the *Region*. As shown in the motivating example, the pump and the space units make up as well as belong to the water supply system.

Definition 5. Group is a collection of *Regions* and *Terminal Groups*, which refers to the collection of related elements in the building.

Definition 6. Individual is a collection of *Basic Units* and *Terminals*, which refers to the single element in *Group*.

The *Region* and *Terminal Group* represent the collection of some elements, while the *Basic Unit* and *Terminal* represent the elements in them. Hence, the collection part is called *Group* and the element part is called *Individual*. These concepts describe the two core abstract parts of the building domain objects.

To construct a *Group* with *Individuals*, some constraint conditions are needed, and they depend on the information model element. One constraint condition is made up of one or more information model elements.

Group and *Individual* correspond to two kinds of Graphic Elements, and another kind of Graphic Elements is Operation. Operation depends on the standard information model because the operation is substantially involved with the reading/writing or calculating of the data in the information model.

Each Graphic Element has one or more configurations, which configure the attributes by changing the data of the standard information model. Therefore, it makes the configuration depend on the information model.

4.2. Programming Model. The programming model is a concrete mapping of the domain model at the implementation level. It represents the hierarchy and connection relationship between the language feature blocks abstracted

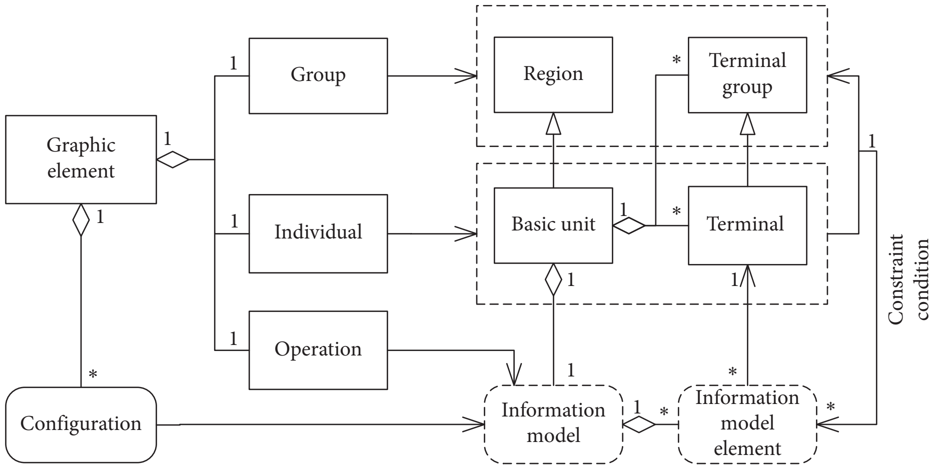


Figure 3: Domain model of graphical programming language and I^{2}B.

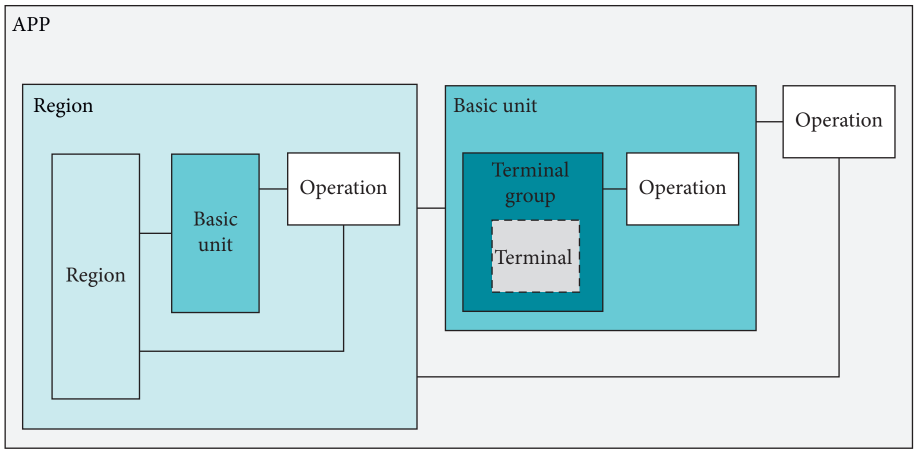


Figure 4: Programming model of the graphical programming language.

by the elements in the field of I^{2}B. The programming model is shown in Figure 4.

An *APP* of I^{2}B can involve multiple *Regions*, *Basic Units*, and Operations, and these elements can be connected to each other.

A *Region* can involve *Regions*, *Basic Unit*s, and Operations, and these elements in a Region can be connected to each other.

A *Basic Unit* can involve *Terminal Groups* and Operations. One *Terminal Group* can contain multiple *Terminals* of the same type but does not contain the operation acting on them.

Here, the Operation element represents a collection of effects on data or processes. The operation of a single terminal device is embodied in read-write data of the *Basic Unit*'s standard information model; thus, no specific Operation element is required to be connected to *Terminals*.

5. Design of Graphical Programming Language

To deal with the developing difficulties of the motivating example in Section 2 and based on the general programming language model refined from the composition of the intelligent building in the physical world, this section presents the design concepts and definitions of the graphic elements to afford the support for the graphical programming language.

5.1. Designing Graphic Elements. As shown in the motivating example, the basic components of I^{2}B are building space units and large electromechanical equipment regarded as intelligent CPN nodes. Each CPN node has a certain computing capacity, including multiple sensors or controlled devices and is bound to one standard information model. Nodes in different functional areas work together to form a complete functional network. As another example, air conditioning units, cooling towers, pumps, exchangers, and end devices make up an air conditioning and cooling function subnet.

The mapping relationships between graphic programming elements and the I^{2}B platform are shown in Figure 5. If the functional subnet is regarded as a community, then each electromechanical equipment and space unit is a member of the community, so the upper layer of function-oriented control is achieved by designing the Community Block (CB)

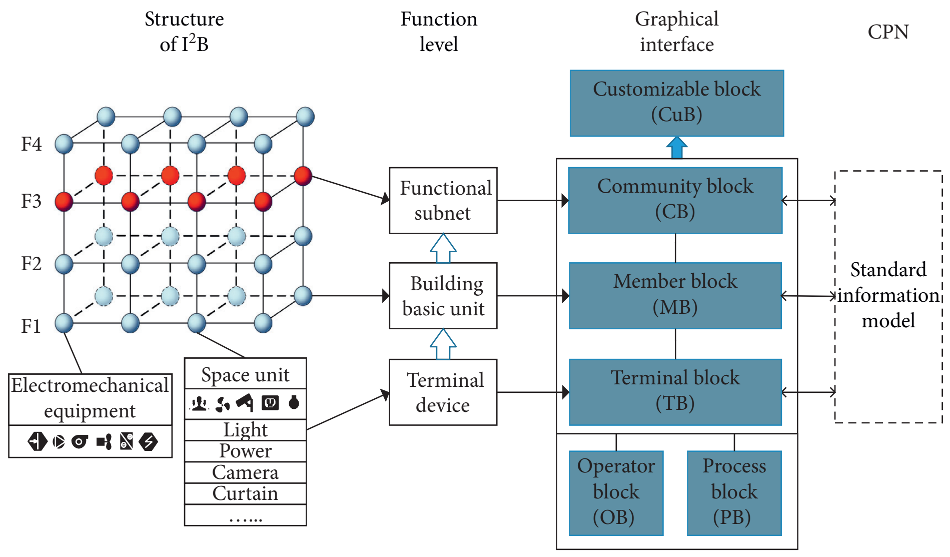


Figure 5: The mapping relationships between graphic programming elements and I^2B.

and the Member Block (MB). The intelligent building is installed and packaged with a large number of sensors in rooms and equipment to complete the collection and measurement of various parameters of the environment and system. The measurement data are maintained in the standard information model in every node. The data can be read and written after calculation and processing to achieve the access and control of terminal devices. The terminal device and its related data in the standard information model are represented as a Terminal Block (TB). The processing of the data is refined into an Operator Block (OB) and the implementation process of the control operation is refined into a Process Block (PB) in order to make the control process explicit and easy to read and understand. Finally, we add a Customizable Block (CuB) that is a complete package, constructed according to functions to isolate programs with different functions.

Corresponding to the motivating example in Section 2, the Member Block is directed against D1, the Terminal Block points at D2, the Community Block is designed on D3, the Operator Block together with the Process Block addresses D4, and the Customizable Block is an addition for practicality and conciseness.

According to the above correspondence, the graphical elements of the graphical programming language are divided into Operator Blocks, Process Blocks, Terminal Blocks, Member Blocks, Community Blocks, and Customizable Blocks according to the actual and functional comparison relationships. Each element block has one basic graphic. To distinguish them intuitively, based on the different functions and structures, the symbols, characters, and device representations are matched in the basic graphics to further characterize and distinguish the modules under the classification of the elements. The developers can perform custom extensions of the graphic elements library on the basis of basic graphics, in combination with the application requirement.

5.2. Implementations of Graphic Programming Elements. Based on the composition of the physical world of the intelligent building, the real-world elements can be mapped to the elements of the programming language, from which the classification of the graphic elements are derived. The classification and examples of our designed graphic elements are shown in Table 2. Each graphic element has a basic shape and can be more specifically described with more symbols.

(1) Community block: corresponding to the subnets that implement a relatively independent and complete function in the decentralized building system

(2) Member block: corresponding to the architectural building space unit and large-scale electromechanical equipment, including six categories and all kinds of member blocks that can be further refined

(3) Variable block: corresponding to the input and output physical quantities of actuators and sensors in intelligent buildings

(4) Operate block: completing the most basic calculations and advanced algorithms

(5) Process block: supporting the programming structure and displaying the process flow of data progression for explicitly presenting the control flow

(6) Customizable block: encapsulating a functional program and supporting text code encapsulation and graphical program encapsulation

5.3. Formally Defining the Graphic Programming Elements. Table 2 presents the basic graphical definition of graphic elements, which is for an intuitive understanding and implementation. To facilitate the code implementation, the formal definitions of graphic elements are needed and can be defined in the form of a multiple array.

Table 2: Classification and examples of graphic elements.

Element category	Content	Basic graphics	Examples		
Member block	Six categories: Space units, HVAC, water system, power distribution, firefighting, and security, which can be further refined based on devices	□	AHU	Pump	Space unit
Community block	With two nodes: Starting community block and ending community block		Starting community block	Ending community block	Water system
Terminal block	Terminal devices, actuators, sensors, and variables	⬡	Pressure	Temperature	Group of lights
Operate block	Mathematical operations, logic operations, network operations, and parameters	○	Sum	Logical operations AND	Maximize
Process block	Delay, loop, select, jump, time count, number count, trigger, etc.	◇	Trigger	Selection	Timer
Customizable block	Program encapsulation		Customizable SUM	Customizable pressure increase	Customizable attemperation

Obviously, each element has an original fundamental definition part, which is the basic component of the element. Each device in the intelligent building system is linked to the standard information model; therefore, the standard information model should be included in the definition of an element so that the device information can be simply read and manipulated via the elements.

Definition 7. Base =<*N, A, I, L, C, F*> is the fundamental elements group, where

(i) *N* means Name for identifying the graphic element's name

(ii) *A* means Attribute for identifying attribute types such as element type, status, and data type

(iii) *I* means Image information that is used to store graphic icons and displays graphics information such as their shape, color, and size

(iv) *L* means Link for identifying the connection status and connection objects

(v) *C* means Code that corresponds to the code calling method of the graphic element

(vi) *F* means Flag and stores status flags such as selection, connection, and conversion, to respond to state changes of elements

5.3.1. Operate Block (OB). Operate Block completes the most basic calculations such as mathematical operations, logical operations, and the parameters involved in the operation and supports the extended encapsulation of custom algorithms for distributed network computing and advanced algorithms. Its main content includes

Mathematical operations: add, subtract, multiply, divide, power, integral, differential, comparison, random, and approximation

Logical operations: AND, OR, NOT, and XOR

Network operations: synchronous send and synchronous receive

Parameters: constants, variables, and type conversions

Custom function: advanced algorithms such as maxima, minimum, set operations, spanning tree, iteration, matrix calculation, steepest descent method, Newton method, genetic algorithm, and neuron algorithm

Among them, the network operation block is designed for parallel network computing and is used to exchange information between members in the community. When synchronously sending, each member sends the data to all neighbors at the same time and does not perform other actions. When synchronously receiving, each member simultaneously receives all data from neighbors to the local. Both of these processes are only for neighbors, that is, they only exchange data with members who are actually connected to themselves. This operation can effectively complete the data exchange during parallel computing without explicitly sending and receiving objects, thereby achieving efficient and simple programming.

Definition 8. OB =<Base, Ope> is an Operate Block defined by a multiple array, where

(i) *Base* refers to the fundamental elements group

(ii) *Ope* refers to the specific operation code of calculation or algorithm

5.3.2. Process Block (PB). Process Block intuitively supports system programming and displays the process flow of data progression for explicitly presenting the control flow, including delay, loop, loop stop, interrupt, select, jump, time count, and number count. The block is the key to demonstrating event triggering, with which the program can carry out different processes.

Definition 9. PB =<Base, Pro> is a Process Block defined by a multiple array, where

(i) *Base* refers to the fundamental elements group

(ii) *Pro* refers to the specific operation code of process control

5.3.3. Terminal Block (TB). Termianl Block achieves the reading and control of the terminal devices in the basic units. It represents two types of devices: the sensing device for collecting external environmental parameters or state changes in the system, including various sensors such as temperature, humidity, pressure, illumination, flow, gas composition and concentration, smoke, and switches and buttons, and the controlled device for performing the action or state switching, including controlled switches, and various types of single electrical devices such as electric lights, audio, and display screens. These devices are present inside the space unit or electromechanical device.

Definition 10. TB =<Base, SIME> is a Terminal Block defined by a multiple array, where

(i) *Base* refers to the fundamental elements group

(ii) *SIME* refers to the mapping of a standard information model's element in the CPN corresponding to the Member Block

5.3.4. Member Block (MB). Member Block corresponds to the building space unit and large-scale electromechanical equipment in I^2B and to the basic division of the CPN. It mainly includes six categories: air conditioning, water system, power distribution, space units, firefighting, and security. All kinds of member blocks can be further refined into more specific member blocks according to different types of devices, and developers can dynamically maintain the graphical primitive library according to the actual operation and maintenance conditions. For example, the air conditioning unit includes the air conditioning box, cold/heat source, water pump, fan, heat exchanger, and controller. The water system includes the water tank, pool, pump, and terminals.

Definition 11. MB =<Base, SIM> is a Member Block defined by a multiple array, where

(i) *Base* refers to the fundamental elements group

(ii) *SIM* refers to the mapping of a standard information model in a CPN corresponding to the Member Block

5.3.5. Community Block (CB). Community Block corresponds to the region and subnets that implement a relatively independent and complete function in the intelligent building system. Each member in the subnet contains some of the same parameters, and subnets are controlled through the data exchange of these parameters. The community can be manually configured by setting its members or configured automatically by identifying and interacting with parameters. The communities that can be automatically organized are functional subnets within the building, such as floors, air conditioning refrigeration, water heating, ventilation, water supply and drainage, power distribution, gas, firefighting, and security.

The block is composed of a pair of symbols, called Starting Community Block and Ending Community Block, which can constitute a closed range to represent a region. The programs in the region are executed in parallel in every member of the community. It is the key to demonstrating parallel tasks and distributed computing.

Definition 12. CB =<Base, SIM>* is a Community Block defined by a multiple array, where

(i) *Base* refers to the fundamental elements group

(ii) SIM^* refers to the mapping of the common part of the standard information model of the members within the community

5.3.6. Customizable Block (CuB). Customizable Block corresponds to a program task with specific functions for visually simplifying the program. It encapsulates a functional program and supports text code encapsulation and graphical program encapsulation.

Definition 13. CuB =<Base, Cus> is a Customizable Block defined by a multiple array, where

(i) *Base* refers to the fundamental elements group

(ii) *Cus* refers to the user-defined program

6. Prototype Support Tool

In this section, we implement a prototype support tool for the graphical programming language. The prototype support tool is realized with the Domain-Specific Language Designer (Visual Studio DSL) of Visual Studio 2017. The Visual Studio DSL is designed specifically for domain-specific development by Microsoft. The DSL tool is included in the Visual Studio SDK and allows developers to design their own special graphical tools with built-in support for models and the relations between models and graphics. It can support model validation, rules, transaction, etc. and allows developers to use some extensions VSX of VS.NET together, such as toolbars and menus, and can be used with T4 (text template transformation toolkit) to generate object code [39]. This section provides the development process of the graphical programming prototype tool of I^2B with Visual Studio DSL, and the processing realizes graphical programming and text code conversion for I^2B. The development of prototype tools mainly includes steps of modeling domain-specific language, setting the interface representation, developing the template engine, and package deployment.

6.1. Modeling Domain-Specific Language. In the process of developing a DSL with Visual Studio DSL tools, the domain model is the core of a DSL, which defines the various concepts and attributes that the language represents and the relationships between them. The domain model is similar to the grammar of a DSL and defines the elements that make up the model and provides the rules that tie these elements together. In the "DslDefinition.dsl" file, we use the UML (unified modeling language) to define the domain model and set the language elements: Operator Blocks, Process Blocks, Terminal Blocks, Member Blocks, Community Blocks, and Customizable Blocks as domain classes, then further set their properties and included domain classes, as well as different domain relationships between domain classes. Part of the domain model settings for this graphical programming language is shown in Figure 6.

6.2. Setting Interface Representation. The interface representations such as graphical symbols, toolboxes, resource managers, and property windows are based on the domain model. They represent the specific information contained in the basic model elements through the designer's user interface. Based on the established domain model, we implement configure Image Shape separately for each domain class that represents a language feature; each image here has been drawn in other tools. Then, the properties IconDecorator and TextDecorator are set for the images, which are used to expand the display of the image and text to make the language graphic elements more readable in the interface.

6.3. Developing Template Engine. Using a graphical language can effectively reduce the burden on developers, but in the final control phase, graphical language cannot be effectively and directly identified and utilized, and must be converted into C language code that has a relatively low level of abstraction and can be recognized by the CPN operating system. Therefore, the conversion of graphical language to textual C language is an important work of the prototype tool. The Visual Studio DSL tool provides a simple and effective graph-to-text conversion method, which is a template-based conversion method. It uses T4 (text template transformation toolkit) to edit and develop the language-specific template that is used as an engine to drive the conversion of graphical language. The key to template-based graph-to-text conversion is the control tags, which are several pairs of prescribed symbols. All elements outside the control tag are output directly to the output file, and the code between the pair of the symbols is executed to generate structural or dynamic behavior. The forms and usage of the template engine's control tags are as follows [40]:

<#@. . .#> Instruction block: provides preprocessing instructions for the engine that executes the text template.

<#. . .#> Standard control block: the content in it is used to control the flow of the template instead of directly as the output of the template. It stores the control statement, that is, the control codes composed of normal C# or VB code.

<#+. . .#> Class property control block: the content in it includes fields, attributes, and inline classes, allowing additional methods to be added to the template, and some highly reusable code is generally stored.

<#=. . .#> Expression control block: calculates the value of the expression contained in it and converts the result into a string for direct output to the generated code.

We input codes in the LanguageSmReport.tt file and save the file to generate a conversion template. Figure 7 shows part of the language conversion template of the graphical programming language in this paper. In the content contained in the instruction block, the *template* instruction specifies that the template is inherited from the standard classes provided by graphical programming language. These classes are used to create text templates for accessing the model file. The *output* instruction specifies that the file type generated by the text template is ".txt," and it can also be a standard file type, such as ".cs," which means that a C# file or other standard file will be generated. The *Language*

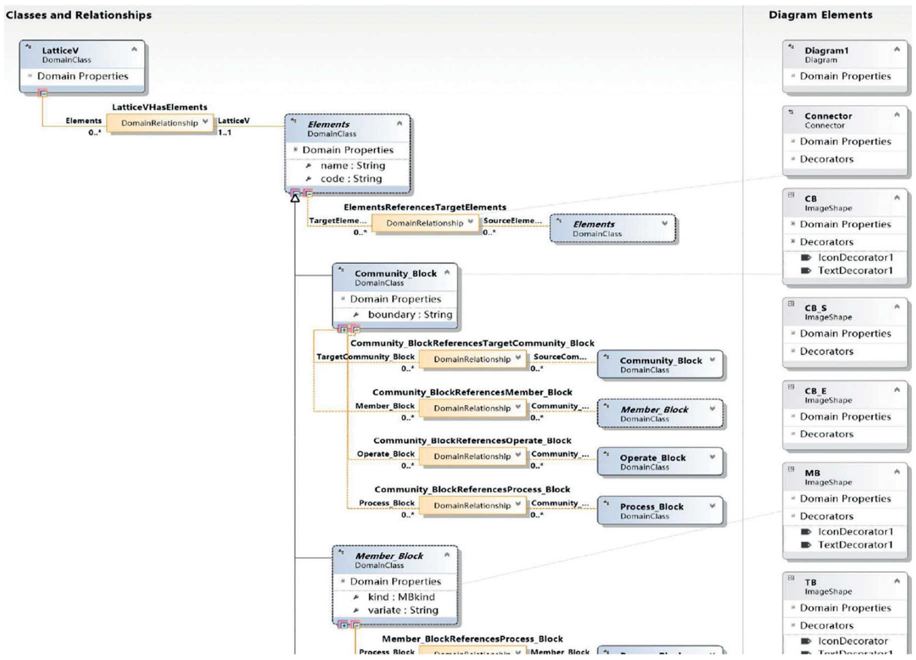
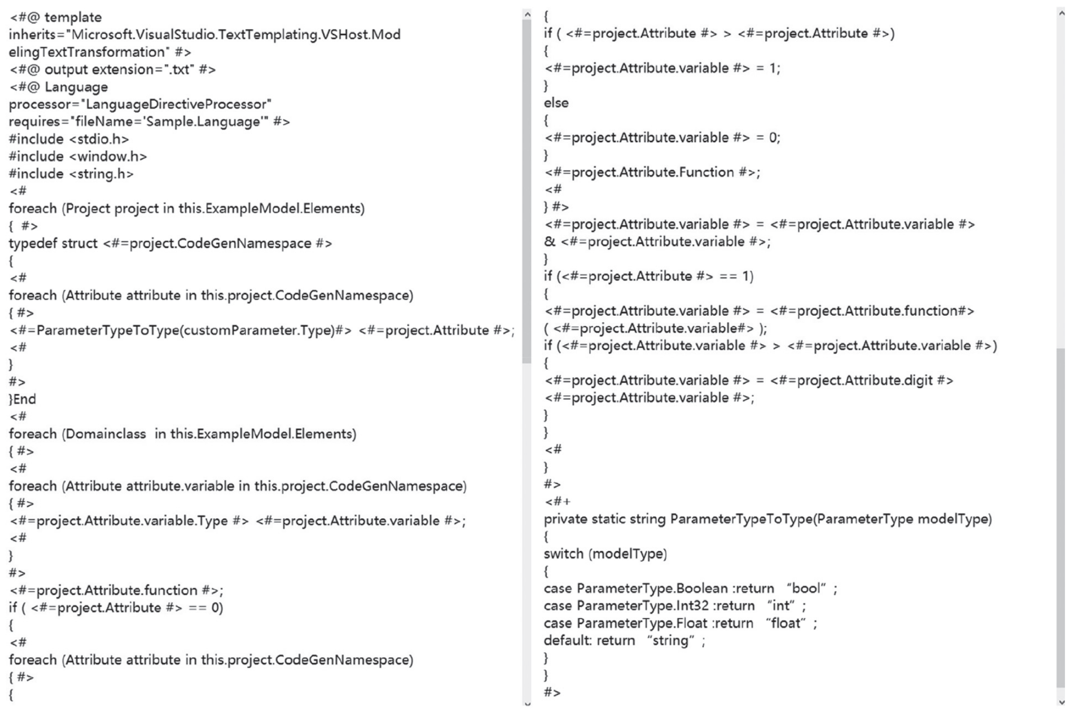


Figure 6: Part of the domain model settings.



```
<#@ template
inherits="Microsoft.VisualStudio.TextTemplating.VSHost.Mod
elingTextTransformation" #>
<#@ output extension=".txt" #>
<#@ Language
processor="LanguageDirectiveProcessor"
requires="fileName='Sample.Language'" #>
#include <stdio.h>
#include <window.h>
#include <string.h>
<#
foreach (Project project in this.ExampleModel.Elements)
{  #>
typedef struct <#=project.CodeGenNamespace #>
{
<#
foreach (Attribute attribute in this.project.CodeGenNamespace)
{ #>
<#=ParameterTypeToType(customParameter.Type)#> <#=project.Attribute #>;
<#
}
#>
}End
<#
foreach (Domainclass  in this.ExampleModel.Elements)
{ #>
<#
foreach (Attribute attribute.variable in this.project.CodeGenNamespace)
{ #>
<#=project.Attribute.variable.Type #> <#=project.Attribute.variable #>;
<#
}
#>
<#=project.Attribute.function #>;
if ( <#=project.Attribute #> == 0)
{
<#
foreach (Attribute attribute in this.project.CodeGenNamespace)
{ #>
{
{
if ( <#=project.Attribute #> > <#=project.Attribute #>)
{
<#=project.Attribute.variable #> = 1;
}
else
{
<#=project.Attribute.variable #> = 0;
}
<#=project.Attribute.Function #>;
<#
} #>
<#=project.Attribute.variable #> = <#=project.Attribute.variable #>
& <#=project.Attribute.variable #>;
}
if (<#=project.Attribute #> == 1)
{
<#=project.Attribute.variable #> = <#=project.Attribute.function#>
( <#=project.Attribute.variable#> );
if (<#=project.Attribute.variable #> > <#=project.Attribute.variable #>)
{
<#=project.Attribute.variable #> = <#=project.Attribute.digit #>
<#=project.Attribute.variable #>;
}
}
<#
}
#>
<#+
private static string ParameterTypeToType(ParameterType modelType)
{
switch (modelType)
{
case ParameterType.Boolean :return  "bool" ;
case ParameterType.Int32 :return  "int" ;
case ParameterType.Float :return  "float" ;
default: return  "string" ;
}
}
#>
```

Figure 7: Part of the language conversion template.

instruction causes the template to load the Sample.Language model file and generate root named *Language* that points to the in-memory model file. Furthermore, statements outside the control tag can be directly output to the output file, such as the *typedef* statement in Figure 7.

Except for instructions and template files that are directly copied to the template output, the rest of the template file consists of three control blocks. Firstly, the content in the standard control block "<#. . .#>" is used to control the flow of the template rather than directly as the output of the template, which introduces control statements into the template. In this template, a *foreach* loop is started, and every time this loop repeats, everything that follows the control block is repeated once until it encounters a control block containing a matching closing brace, i.e., :<# } #>." The loop is controlled by the *Projects* collection, which is extracted from the model file and exposed to the user by the *Language* attribute. Secondly, the name of the class is generated by the expression control block, which is separated by "<#=...#>."The expression control block simply calculates the value of the expression it contains and calls the standard method *ToString()* on the result to convert the result into a string and output it directly to the generated code. Thirdly, the class property control block "<#+...#>" adds methods, attributes, fields, or inline classes to the template, which must be placed at the end of each template file. This is the best way to add reusable code to templates because they can be embedded in parameterized methods, and any local variables they declare will not conflict with other templates. In order to calculate the data type of the variable corresponding to each terminal parameter value in the model, the *switch* statement should be used, but it is very inflexible to repeat this code every time the *switch* statement is needed. Therefore, we use the class property control block to encapsulate this reusable code as a method at the end of the template file. The expression block <#=ParameterTypeToType(ParameterField.Type)#> is used in several places in the template to output the correct type.

Then, using these three kinds of control block, through the loop traversal of the domain classes and transitions in the language program, the output of the class declaration, type definition, and class attributes of the text language are completed. The text template needs to be iterated continuously to be able to go from the smallest generated template to the most feature-rich and mature template. In practice, such an iterative process can also be used in DSL development, by gradually adding more parameterization to make the model more valuable and require less handwritten code.

6.4. Package Deployment. For users of the graphical program editor, a language tool should be portable and scalable windowing software without cumbersome installation steps in order to achieve true friendliness. To achieve this goal, language developers need a method to package their programming language into an installation package. We use the installation project template in the Visual Studio DSL tool to create the ".msi" file installation package from the project of defining the language. WiX (Windows Install XML) is an open source tool for creating the Windows installation package. We write XML scripts via WiX, declare all the components that need to be included in the .msi file, and provide the settings needed to run the .msi installation wizard. The WiX tool can compile the script to the .msi file and generate an installation file that can be installed and deployed [39].

7. Verification

In this section, we verify and evaluate the proposed approach to demonstrate that the graphical programming language for I^2B suffices and facilitates the development of I^2B APPs.

As a new graphical programming language for I^2B, the first question we should answer is whether the language and the prototype tool can support the APP's development, which is the effectiveness verification. Another question is whether the approach improves the productivity, as our purpose is to provide language support for a direct and friendly development of I^2B APPs, which means the efficiency verification.

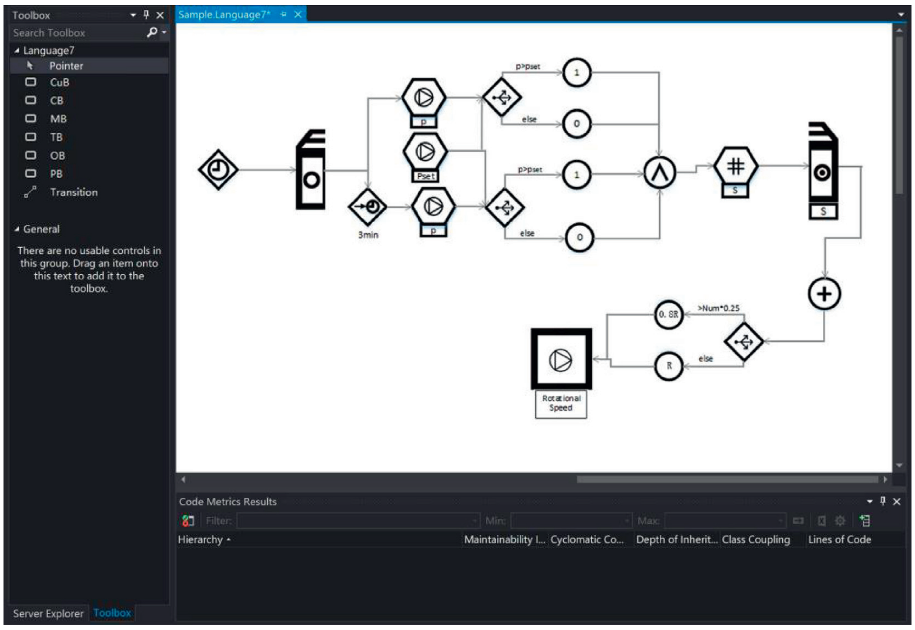
7.1. Effectiveness Verification. In Section 2, we provided a motivating example to inspire the design of the graphical programming language. In this section, we implement this example application with the graphical programming language and the prototype tool. First, we draw the graphical program with the prototyping tool based on the case's control strategy, using the graphical programming language for I^2B. Then, we explain the programming ideas and the procedure of running the graphical program. Finally, we use the prototyping tool to convert the diagram to textual code. The example implementation covers the developing process of I^2B APPs, which can validate the effectiveness of our approach.

7.1.1. Graphical Program. Because of the large scale of the entire water supply network, it is very difficult to program the whole network that contains the pump and all the water supply ends. The distribution of ends in the network is extensive, so it is difficult to employ configure and program one-by-one. However, the quantity of the pressure pumps is very small, and the pumps connect to the water pipe network. Therefore, it is very easy to program the pump and control the system with the community and parallel region.

Figure 8 shows the graphical programming interface and the graphical programming diagram of the pressure reduction control of the building water supply system. The graphical programming interface has a drawing board, a toolbox on the left, and a property window on the bottom. The drawing board is the display area to place graphical elements. The toolbox provides the graphical elements and wiring tool, from which the graphical elements can be dragged to the drawing board. The property window displays the configuration information of the element when clicking on a graphical element and supports modification and configuration of the element's types, parameters, and attributes.

The drawing board of Figure 8 shows the graphical programming diagram of the pressure reduction control of the building's water supply system. The procedure for running the program is as follows:

(1) Set the timer to make the pump periodically initiate water pressure testing of the water pipe network. When the timing triggers, the pump initiates the parallel interaction, constructs the water pipe network community through the configuration and the



There are no usable controls in this group. Drag an item onto this text to add it to the toolbox.

FIGURE 8: The graphical programming interface and the diagram of the graphical platform.

standard data set parameters, and distributes the set value of the water pressure to the members.

(2) All members simultaneously execute the program in the parallel region: compare the water pressure set value and the measured value P and output 1 to the AND block when $P > P\text{set}$; otherwise, output 0. After the timer times out for 3 minutes, the comparison between Pset and P is again triggered, and the result value is output to the AND block for judging whether the water pipe end is continuously in an overpressure state.

(3) After the AND block obtains the two values, the AND result is output to the variable S. If the pipe end sustains overpressure, $S = 1$; otherwise, $S = 0$.

(4) The end community block aggregates the value of the variable S of all members into a summation block to judge the total overpressure degree with the value of the sum.

(5) If it is judged that the sum value is greater than the specified overpressure limit, the rotation speed of the pump motor is adjusted to 0.8 times that of the original; otherwise, the speed is maintained and a waiting period for the next program execution proceeds. The total overpressure degree of the water pipe network is converted to the correction value of the pump rotating speed, and the pressure reduction control of the water supply system is realized. The pressure increase control of the water supply system is similar.

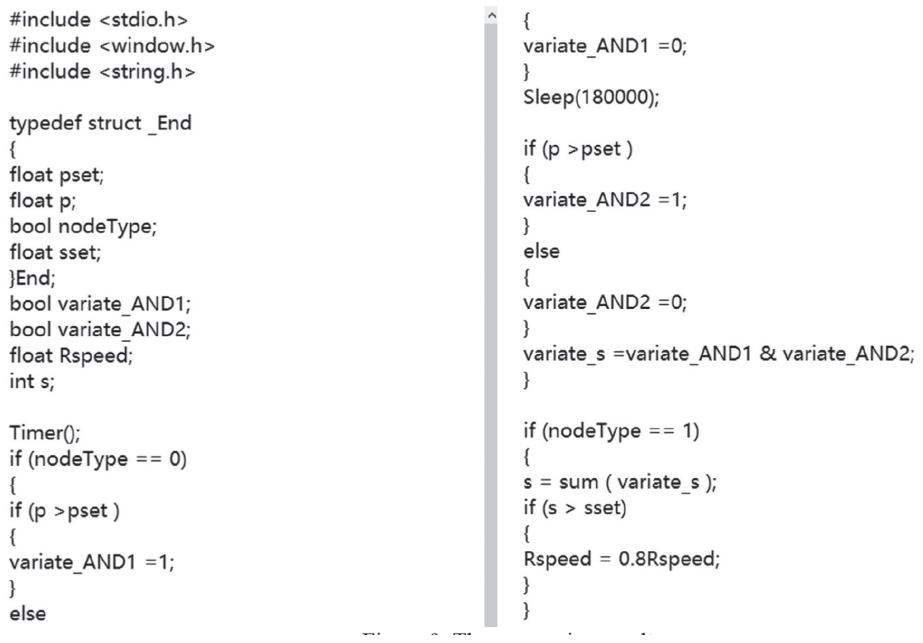
7.1.2. Graph-to-Text Conversion. The automatic conversion of the graphical program to the textual program can be achieved via the conversion template. Right-click on the template file LanguageSmReport.tt and click Run Custom File to generate a .txt file containing the target language of the program according to the control and constraint programs in the template. The conversion result is shown in Figure 9, where the variable *nodeType* corresponds to the standard information model parameter and represents the type of member block. When *nodeType* is equal to 0, it represents the end member block of the water pipe, and when *nodeType* is equal to 1, it represents the water pump member block. *Sum* is a function that sums the variables *variate_s* of all members in the community block.

7.2. Efficiency Verification. To verify whether the proposed programming language can provide the user with simple and friendly support for the development of I^{2}B APPs, we conduct comparative experiments to evaluate the efficiency in the actual use process.

7.2.1. Experimental Methods. We adopt the method of contrast experiments. The experimental participants are 4 graduate students from our research group. They have basic programming experience and had a certain understanding of I^{2}B operation and control methods. Participants are randomly divided into two groups, which program using C language and the proposed programming language. First, each group receives training on relevant programming languages and application tasks, and then they are required to independently develop two typical APPs for I^{2}B. We measure the development and use efficiency of the programming language through training time, developing time, and completion degree.

7.2.2. Experimental Tasks

APP1 (light control via number of people): turn on the lights of the lab when the total number of people on the entire floor is more than 15; otherwise, turn off the lights.



```
#include <stdio.h>
#include <window.h>
#include <string.h>

typedef struct _End
{
float pset;
float p;
bool nodeType;
float sset;
}End;
bool variate_AND1;
bool variate_AND2;
float Rspeed;
int s;

Timer();
if (nodeType == 0)
{
if (p >pset )
{
variate_AND1 =1;
}
else
{
variate_AND1 =0;
}
Sleep(180000);

if (p >pset )
{
variate_AND2 =1;
}
else
{
variate_AND2 =0;
}
variate_s =variate_AND1 & variate_AND2;
}

if (nodeType == 1)
{
s = sum ( variate_s );
if (s > sset)
{
Rspeed = 0.8Rspeed;
}
}
```

Figure 9: The conversion result.

APP2 (AHU control via total air volume): first, determine the local demand air volume based on the set temperature, measured temperature of each room, and the known function f1, and then use the sum of the demand air volumes of all rooms and the known function f2 to adjust the AHU fan; in addition, readjust the fan rotating speed whenever the set temperature changes.

7.2.3. *Experimental Metrics*

Training time (TT): participants' learning time for the programming language and the two experimental tasks before programming, representing the learning costs and acceptance degree of the languages.

Programming time (PT): the time it takes for participants to develop an application using the programming languages, representing the use cost, acceptance degree, and efficiency of the languages. The prototype supporting tool is not a mature practical product and the time it takes for prototype tools to convert graphics to text cannot reflect the real running efficiency; thus, the conversion time is not counted in PT. PT is only the time it takes to build a graphics program using the graphical programming language and to write text code in C.

Completion degree (CD): the general correctness degree of the C code and the final text code converted by the prototype tool through checking the logic, syntax, and function. When the programming goals are clear and consistent, the lower the CD is, the more difficult it is for the language users to accept and use the language, so this metric is used to help represent the acceptance degree of the programming languages.

7.2.4. Experimental Result. The results of the experiments are shown in Table 3.

Although using the proposed programming language requires a certain amount of training time, its programming time is obviously shorter than the general language C.

The proposed programming language achieves a better efficiency performance in APP2 than in APP1. This finding occurs because APP2 is more complex than APP1 and contains more control mechanisms and language elements, and the proposed language can describe application tasks more concisely and efficiently with graphic elements that are specifically designed for I^2B; in particular, Community Blocks can easily realize collective parallel computing, which can greatly shorten the programming time.

The CD of the two groups represents that both programming languages can basically complete the correct programming of I^2B APPs. However, the CD of group A is generally lower than that of group B, which is limited by the defects of the prototype tool's conversion template.

7.3. Summary and Discussion. Based on the analysis of the two verification results, the following conclusions can be drawn:

(1) The proposed graphical programming language can realize the development of I^2B APPs and realize the conversion to the text language through the prototype support tool, which demonstrates the effectiveness of our approach;

(2) Compared with the general language, the proposed language can improve users' programming efficiency and has better friendliness and usability. In comparison, our approach can realize shorter development time with an acceptable training duration, and the completion degree is similar to that of C language;

Table 3: The results of the comparative experiments.

Participant		TT (min)	APP1		APP2	
			PT (min)	CD (%)	PT (min)	CD (%)
Group A (ours)	P1	15	6	96	9	94
	P2		5	96	9	94
Group B (C)	P3	8	9	97	17	96
	P4		9	98	18	97

(3) The advantages of our approach over the general programming language emerge from the more complex control tasks. Because the language is specifically designed for I^{2}B, its dedicated elements and meanings can effectively reduce the threshold of the development of I^{2}B APPs, providing users with a direct and friendly programming method;

(4) After using the proposed programming language, participants express that the graphical programming language specific to the I^{2}B domain is relatively easy to learn and use, and the staff in the intelligent building field can participate in programming and better apply their expertise.

The prototype support tool implements the general function of graphical program development and graph-text conversion, but the research of the prototype tool still requires considerable work, and some deficiencies need to be addressed:

(1) The toolbox of graphic elements in the prototype tool is all from an artificial addition. The considered types of elements have not covered all the domain classes of the entire intelligent building;

(2) The precision of the graphics conversion template engine setting is insufficient. After debugging, the basic textual conversion for the case can be realized, but it cannot support all the graphical program conversions perfectly;

(3) C programming language is the target language of textual conversion. The parallelism of the conversion result language needs to be improved because of the conversion template engine setting and the target language selection.

8. Conclusions

In this paper, we have proposed a graphical programming language for programming and development of the Insect Intelligent Building. We have analyzed the domain characteristics and the language model of I^{2}B. Moreover, we have specifically discussed the design motivation and definition of the graphic elements, combined with the characteristics and background of the system and related developers. The proposed graphical programming language can reduce the difficulty and learning cost of programming I^{2}B APPs and provide the support of parallel group control with a set of domain-specific and friendly programming methods.

Moreover, we have proposed a prototype tool for the development of I^{2}B using the graphical programming language and presented the implementation process of the prototype tool. Finally, we conducted verification experiments. On the one hand, the specific programming example reveals the diagram and implication of the graphical program and implements the graph-to-text conversion with the prototype tool, which indicates the applicability of the graphical programming language and the feasibility of the prototype tool. On the other hand, comparative experiments show the better efficiency performance of our work.

Actually, the proposed graphical programming language in this paper is only a prototype design oriented to I^{2}B and its developers, and not comprehensive and sufficiently appropriate to support all control methods in intelligent buildings. In addition, the prototype tool is just in the preliminary stage, and its evolution to an engineering development environment and a practical product is a rather long process. In the future, we will continue to enrich the expression ability of the graphical programming language and strengthen support for complex algorithms such as network computing. On the contrary, in response to the deficiencies of the prototype tool, we will further improve the completeness of the graphics toolbox and the universality of the conversion template and add the capability of generating a more advanced programming language.

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 supported by the National Key Research and Development Project of China (new generation intelligent building platform techniques) (2017YFC0704100)

References

[1] J. K. W. Wong, H. Li, and S. W. Wang, “Intelligent building research: a review,” *Automation in Construction*, vol. 14, no. 1, pp. 143–159, 2005.

[2] Q. Zhao and Z. Jiang, “Insect intelligent building (I2B): 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.

[3] M. F. Sanner, “Python: a programming language for software integration and development,” *Journal of Molecular Graphics and Modelling*, vol. 17, no. 1, pp. 57–61, 1999.

[4] B. Eckel, *Thinking in Java*, Prentice Hall Professional, Upper Saddle River, NJ, USA, 2003.

[5] R. Ierusalimschy, L. H. De Figueiredo, and W. Celes, “The evolution of Lua,” in *Proceedings of the History of Programming Languages III*, pp. 2-1–2-26, ACM, San Diego, CA, USA, June 2007.

[6] D. Fogli, R. Lanzilotti, and A. Piccinno, “End-user development tools for the smart home: a systematic literature review,”

in *Distributed, Ambient and Pervasive Interactions*, N. Streitz and P. Markopoulos, Eds., pp. 69–79, Springer International Publishing, Cham, Switzerland, 2016.

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

[8] R. Morgan, G. Grossmann, and M. Stumptner, “VizDSL: towards a graphical visualisation language for enterprise systems interoperability,” in *Proceedings of the International Symposium on Big Data Visual Analytics (BDVA)*, pp. 1–8, IEEE, Adelaide, Australia, November 2017.

[9] M. Almorsy, J. Grundy, R. Sadus, W. van Straten, D. G. Barnes, and O. Kaluza, “A suite of domain-specific visual languages for scientific software application modelling,” in *Proceedings of the 2013 IEEE Symposium on Visual Languages and Human Centric Computing*, pp. 91–94, IEEE, San Jose, CA, USA, September 2013.

[10] C. Rieger, “Business apps with MAML: a model-driven approach to process-oriented mobile app development,” in *Proceedings of the Symposium on Applied Computing*, pp. 1599–1606, ACM, Marrakech, Morocco, April 2017.

[11] M. Resnick, J. Maloney, A. Monroy-Hernández et al., “Scratch,” *Communications of the ACM*, vol. 52, no. 11, pp. 60–67, 2009.

[12] B. Yang, J. Li, and Q. Zhang, “G language based design of virtual experiment platform for communication with measurement and control,” *Procedia Engineering*, vol. 29, pp. 1549–1553, 2012.

[13] S. Zhao, Q. Yang, J. Xing, and G. Xue, “Graphical programming language design for decentralized building intelligent system,” in *Advancements in Smart City and Intelligent Building, ICSCIB 2018. Advances in Intelligent Systems and Computing*, Q. Fang, Q. Zhu, and F. Qiao, Eds., vol. 890, Springer, Singapore.

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

[15] 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.

[16] Z. Jiang, Y. Dai, and Y. Jiang, “Swarm intelligent building automation system,” *Journal of HV&AC*, vol. 49, pp. 2–17, 2019, in Chinese.

[17] Z. Jiang, “Distributed computing network system and computing node used therefor,” U.S. Patent Application 15/740,146, 2018.

[18] D. He, Q. Xiong, X. Zhang, Y. Dai, and Z. Jiang, “A decentralized, flat-structured control system for chiller plants,” *Applied Sciences*, vol. 9, no. 22, p. 4811, 2019.

[19] S. Wang, J. Xing, Z. Jiang, and J. Li, “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.

[20] “Google for Education: Blockly, Google developers,” 2019, https://developers.google.cn/blockly/guides/get-started/web.

[21] A. Kay, *Squeak Etoys Authoring & Media*, Viewpoints Research Institute, Glendale, CA, USA, 2005.

[22] M. Gasperi and P. Hurbain, *Extreme NXT: Extending the LEGO MINDSTORMS NXT to the Next Level*, Apress, New York, NY, USA, 2010.

[23] International Electrotechnical Commission, *IEC61131-3: Programmable Controllers–Part 3: Programming Languages*, International Electrotechnical Commission, Geneva, Switzerland, 2nd edition, 2003.

[24] S. Campanelli, P. Foglia, and C. A. Prete, “An architecture to integrate IEC 61131-3 systems in an IEC 61499 distributed solution,” *Computers in Industry*, vol. 72, pp. 47–67, 2015.

[25] V. Vyatkin, “IEC 61499 as enabler of distributed and intelligent automation: state-of-the-art review,” *IEEE Transactions on Industrial Informatics*, vol. 7, no. 4, pp. 768–781, 2011.

[26] N. Khan, A. K. Ali, M. J. Skibniewski, D. Y. Lee, and C. Park, “Excavation safety modeling approach using BIM and VPL,” *Advances in Civil Engineering*, vol. 2019, Article ID 1515808, 15 pages, 2019.

[27] T. W. Kang, “Object composite query method using IFC and LandXML based on BIM linkage model,” *Automation in Construction*, vol. 76, pp. 14–23, 2017.

[28] G. Desolda, C. Ardito, and M. Matera, “Empowering end users to customize their smart environments: model, composition paradigms, and domain-specific tools,” *ACM Transactions on Computer-Human Interaction*, vol. 24, no. 2, p. 12, 2017.

[29] D. Caivano, F. Cassano, D. Fogli, R. Lanzilotti, and A. Piccinno, “We@Home: a gamified application for collaboratively managing a smart home,” in *Ambient Intelligence—Software and Applications (ISAmI 2017)*, J. F. De Paz, V. Julián, G. Villarrubia, G. Marreiros, and P. Novais, Eds., pp. 79–86, Springer International Publishing, Cham, Switzerland, 2017.

[30] C. Ardito, P. Buono, M. F. Costabile et al., “Enabling end users to define the behavior of smart objects in AAL environments,” in *Ambient Assisted Living*, A. Leone, A. Caroppo, G. Rescio, G. Diraco, and P. Siciliano, Eds., pp. 95–103, Springer International Publishing, Cham, Switzerland, 2018.

[31] Z. Chen, F. Wang, and R. Ma, “Questionnaire analysis of public building intelligent system,” *Intelligent Buildings International*, vol. 3, no. 175, pp. 13–18, 2015.

[32] Z. Chen, F. Wang, and R. Ma, “Investigation and analysis of the status quo of public building intelligent system,” *Intelligent Buildings International*, vol. 3, pp. 26–33, 2015.

[33] F. Wang and Z. Cheng, “Research on investment benefit evaluation method of building intelligent system,” *Building Science*, vol. 12, 2016.

[34] Z. Yu and H. Li, “Research progress in building automation technology,” *Building Science*, vol. 29, no. 10, pp. 106–113, 2013.

[35] China Academy of Building Research, *Investigation of Requirements of Intelligent Building Project*China Academy of Building Research, Beijing China, 2018, https://www.wjx.cn/newwjx/manage/myquestionnaires.aspx.

[36] S. S. Shrestha, “Performance evaluation of carbon-dioxide sensors used in building HVAC applications,” Doctorial Dissertation, Iowa State University, Ames, IA, USA, 2009, https://lib.dr.iastate.edu/etd/10507/.

[37] B. W. Boehm, “Software engineering,” *IEEE Transactions on Computers*, vol. 25, no. 12, pp. 1226–1241, 2006.

[38] M. A. V. J. Muthugala, P. H. D. A. S. Srimal, and A. G. B. P. Jayasekara, “Enhancing interpretation of ambiguous voice instructions based on the environment and the user’s intention for improved human-friendly robot navigation,” *Applied Sciences*, vol. 7, no. 8, 2017.

[39] S. Cook, G. Jones, S. Kent, and A. C. Wills, *Domain-Specific Development with Visual Studio DSL Tools*, Addison-Wesley Professional, Boston, MA, USA, 2007.

[40] T. Kosar, M. Mernik, and P. E. M. Lopez, “Experiences on DSL tools for visual studio,” in *Proceedings ofd the 29th International Conference on Information Technology Interfaces*, pp. 753–758, IEEE, Cavtat, Croatia, June 2007.