Hindawi
Scientific Programming
Volume 2020, Article ID 8887588, 26 pages
https://doi.org/10.1155/2020/8887588

Research Article

Touch: A Textual Programming Language for Developing APPs of Insect Intelligent Building

Wenjie Chen,[1] Qiliang Yang,[1] Ziyan Jiang,[2] Jianchun Xing,[1] Qianchuan Zhao,[3] Qizhen Zhou,[1] and Deshuai Han[4]

[1]*College of Defense Engineering, Army Engineering University of PLA, Nanjing 210007, China*
[2]*Building Energy Research Center, Tsinghua University, Beijing 100084, China*
[3]*Department of Automation, Tsinghua University, Beijing 100084, China*
[4]*College of Combat Support, Rocket Force University of Engineering, Xi'an 710000, China*

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

Received 26 May 2020; Revised 28 July 2020; Accepted 1 September 2020; Published 18 September 2020

Academic Editor: Francisco Ortin

Insect intelligent building (I^2B) is a novel decentralized, flat-structured intelligent building platform with excellent flexibility and scalability. I^2B allows users to develop applications that include control strategies for efficiently managing and controlling buildings. However, developing I^2B APPs (applications) is considered a challenging and complex task due to the complex structural features and parallel computing models of the I^2B platform. Existing studies have been shown to encounter difficulty in supporting a high degree of abstraction and in allowing users to define control scenarios in a concise and comprehensible way. This paper aims to facilitate the development of such applications and to reduce the programming difficulty. We propose Touch, a textual domain-specific language (DSL) that provides a high-level abstraction of I^2B APPs. Specifically, we first establish the conceptual programming architecture of the I^2B APP, making the application more intuitive by abstracting different levels of physical entities in I^2B. Then, we present special language elements to effectively support the parallel computing model of the I^2B platform and provide a formal definition of the concrete Touch syntax. We also implement supporting tools for Touch, including a development environment as well as target code generation. Finally, we present experimental results to demonstrate the effectiveness and efficiency of Touch.

1. Introduction

Intelligent building (IB) and building automation (BA) systems play an important role in most complex modern buildings [1]. The current building control system is centralized, and configuring it during the deployment process requires considerable manpower. However, as the complexity of building system structures increases, it gradually becomes difficult to upgrade and reconstruct, implement cross-system functions, and apply software customizations to the existing centralized control system [2, 3].

To solve the above problems, a novel decentralized, flat-structured intelligent building control system called insect intelligent building (I^2B) [4] has been proposed that regards the entire building as a network with a parallel computing model composed of a series of computing process nodes (CPNs) [5]. The I^2B system changes the design pattern of the centralized system architecture of traditional intelligent buildings and has the following characteristics: efficient sharing of underlying information, self-identification, self-organization, self-coordination, easy operation, easy transformation, and easy expansion [6]. Based on a standardized network platform and a decentralized parallel computing model, the I^2B system allows users to develop applications (APPs) that include building control strategies and download them to the CPN platform to execute and implement various control management tasks, for example, chiller plant control [7] or sensor fault diagnosis [8]. This approach decouples the application software from the building and achieves building application software universality.

However, developing an I^2B APP is quite challenging: (i) distributed system architecture requires applications to have good portability and scalability; (ii) data parallelism increases the complexity of user-designed programs; and (iii) as users' personalized needs change and grow, a concise, friendly, and intuitive programming language should be designed that can be used to develop APPs efficiently.

General application development methods based on general-purpose programming languages (GPLs) such as C language and Python do not consider the task of domain-oriented incremental development [9], and the resulting programs are often highly complex, with poor portability and scalability; thus, they cannot support the development and maintenance of I^2B applications in a flexible way. Using a domain-specific language (DSL) [10, 11] is an alternative approach to I^2B APP development. A DSL provides a development tool tailored toward a particular application domain at a higher level. In some specific domains, DSLs can reduce the difficulty of programming and improve the productivity of development [12–20]. For example, DSLs applied to parallel computing, such as Spar [21], Smartlog [22], and ParDSL [23], effectively support the paradigm of parallel programming but the developer needs to specify the exact communication address during the programming process, which results in poor program portability and robustness. Although some Internet of Things (IoT) programming methods [24–26] and multiagent programming languages [27, 28] support network-parallel programming and addressless communication, these languages do not consider the domain characteristics of I^2B, causing the complex interactions among nodes to be overfocused and making programming difficult. In our previous work [29], a programming model called INR (i.e., Individual, Neighborhood, and Region) is established for I^2B APP development and to motivate programming language design, which abstracted and decomposed the actual application tasks of I^2B into software entities in application software. The INR programming model eliminates dependencies between the application program and the CPN communication address. Its tag-based programming and clustering operation characteristics make the APP universal and parallel. Although INR is quite descriptive, the time-varying characteristics of state parameters in buildings are not considered, which limits INR's applicability to wider application scenarios. In addition, the mailbox method used for information exchange between CPNs makes the APP development process overemphasize sending and receiving displayed messages, resulting in a more redundant program. Furthermore, INR is relatively complicated, which makes the developed programs less intuitive. More critically, the INR programming model only provides the corresponding programming abstraction and programming mechanism and does not provide a way to display language elements and syntax; therefore, it cannot be used directly for I^2B APP development.

To meet these challenges, this paper provides substantial improvements over our previous study [29] and presents Touch, a DSL for I^2B APP development, which can adapt to the unique features of I^2B such as decentralized, distributed, and parallel and reduce the difficulty of I^2B APP development. Touch comprehensively considers the time-varying characteristics of state parameters in the standard information model (SIM), simplifies the information exchange method between CPNs, and can be directly applied to I^2B APP development. An APP developed with Touch has good portability and extensibility, and the developed programs are more concise and intuitive. The main contributions of this paper are as follows:

(i) We propose a high-level programming conceptual architecture for Touch that describes abstract entities in I^2B APPs. Touch mainly includes three key programming concepts: *Basic Unit*, *Neighborhood*, and *Domain*.

(ii) We design a series of Touch language elements rooted in building domain requirements, CPN computing platform, and controlling program, including *Special Data Type*, *Time Offset*, *Location*, and *Event*, to facilitate the development of such applications.

(iii) We implement a tool for Touch that supports I^2B APP development that provides a friendly GUI for editing and testing applications in the understandable Touch language. It also automatically converts Touch-based applications into target code that can be executed on a CPN.

(iv) Experiments are conducted to evaluate the effectiveness and efficiency of Touch. The proposed Touch is applied to the personnel evacuation application to prove its effectiveness. Moreover, a subject-based empirical evaluation is also conducted to evaluate the efficiency of the proposed Touch. The results show that Touch can considerably facilitate I^2B APP development.

The remainder of this paper is organized as follows. In Section 2, we discuss the related work. In Section 3, we provide an introduction to I^2B system architecture, a description of a typical case involving building personnel evacuation, and a domain analysis for I^2B. The overall framework for Touch language development and the programming conceptual architecture of Touch is proposed in Sections 4 and 5, respectively, and the key language elements and concrete syntax of Touch are presented in Section 6. Then, we introduce the implementation of the tools for Touch in Section 7 and evaluate Touch using the personnel evacuation example in Section 8. Section 9 provides a summary and concludes the paper.

2. Related Work

Since the emergence of I^2B, considerable research efforts have been devoted to the subject of I^2B application tasks, and several distributed optimization methods have been proposed. Dai et al. [30] introduced a new decentralized method to solve the optimal distribution problem in heating, ventilation, and air conditioning (HVAC) systems. Yu et al. [31] presented a distributed control structure for parallel pumps

in which the main controller is replaced by several independent distributed controllers that work together to handle global tasks. Zhao et al. [32] investigated a distributed optimal control method for peer-to-peer control of parallel pumps in HVAC systems. Wang et al. [33] proposed a decentralized method for optimal load distribution in an isolated power system that distributes economic load dispatching to each intelligent distributed generator without the need to monitor the host. These methods can improve the efficiency of building control. However, they all solve a specific task. Due to the diversity and difference of I^2B application tasks, it is difficult for works such as these to provide a general development method.

In recent years, DSL-based application development methods for distributed, parallel systems have been extensively investigated. Membarth et al. [34] proposed a heterogeneous image processing acceleration (HIPAcc) framework, which included a DSL for image processing that captures domain-specific algorithm characteristics. They also proposed a source-to-source compiler for generating efficient parallel implementations. Janjic et al. [35] presented a refactored pattern language (RPL) used to represent the parallel structure of an application and capture the properties of its additional features. Kindlmann et al. [36] designed Diderot, a parallel DSL that allows mathematical concepts to be expressed directly in the code and supports parallel execution on modern multicore processors and GPUs. Sredojević et al. [28] introduced an extension of the agent-oriented DSL called ALAS to support distributed nonaxiomatic reasoning for developing specific types of intelligent agents. These programming languages effectively describe their intended specific domain applications, but they require users to be familiar with the underlying structure of the system and directly specify the communication address, which is difficult to apply in I^2B since it requires addressless communication.

DSL in civil engineering has been the focus of many efforts. Park et al. [37, 38] introduced a DSL called KBimCode, which translates the building permit-related regulations in the Korean Building Act into computable rules, reflecting the obvious characteristics of the Korean Building Act sentences. Alves et al. [39] designed the Building Information Modeling Sensor Language (BIMSL) to help create services and application software that combines sensor data with BIM models. Lee et al. [40] presented the BERA (Building Environment Rules and Analysis) language for constructing building information models (BIM) in an intuitive way, improving the efficiency of defining, analyzing, and checking BIM rules. Grabska et al. [41] proposed two new visual languages for creating multistorey building design schemes, which support designers to design 2D floor layouts and 3D building structures, respectively. However, these methods are not suitable for the distributed and parallel I^2B network architecture, and the domain characteristics are quite different from I^2B, so it is difficult to apply them to I^2B APP development.

Many studies have also been conducted regarding the Internet of Things (IoT) programming. Nguyen et al. [42] proposed a framework for IoT application development that separates the programming language and its execution model from the underlying operating system and hardware. Riliskis et al. [43] presented Ravel, a three-tier architecture-based IoT application programming framework that allows developers to explicitly write a three-tier architecture using models, views, and controllers like building web applications. Verriet et al. [44] introduced a virtual prototype method for IoT applications and similar distributed control applications that has clear application specifications and can detect and correct behavior errors and performance issues during the early stage of development. Salman and Al-Yasiri [45] presented the SenNet language for developing wireless sensor network applications that use macroprogramming concepts to abstract the complex tasks and operations required by developers. These programming languages support network-parallel programming and addressless communication but overfocus on the complex interactions between nodes and do not consider the domain characteristics of I^2B, making programming difficult.

In our previous work [29], a programming model called INR was proposed to effectively promote the development of I^2B APPs, which established programming abstractions such as "Individual," "Neighborhood," and "Region" and the "Tag-Based Programming and Clustering Operation" programming mechanism. Although the INR is very descriptive for I^2B application tasks, it does not consider the time-varying characteristics of the state parameters in the SIM, its proposed CPN interaction mode of "mailbox" is relatively complicated, and more importantly, it does not provide the language elements, syntax, and development tools of a programming language, so it cannot be directly used in the development of I^2B APP.

3. Background and Domain Analysis

An overview of I^2B is provided to facilitate an understanding of Touch. The organization structure and operation mechanism of I^2B are briefly introduced. Then, a motivating example is given to demonstrate the characteristics of I^2B APPs. Furthermore, we conduct a domain analysis of I^2B to provide information for Touch language design.

3.1. I^2B System Architecture. I^2B regards the entire building as a decentralized, flat-structured network of building space units and large electromechanical equipment [4]. Each building space unit and electromechanical equipment is controlled by a smart node, called a computing process node (CPN) [5], which is embedded with a microprocessor that has comprehensive information processing and calculating capabilities. In addition, each CPN is equipped with a communication port to support communication with the local drive controller unit (DCU). The DCU manages the local sensors and equipment controllers that can monitor all the information of the building space unit and large electromechanical equipment, for example, equipment operating information, physical parameters, operating feedback, set values, etc. Due to the communication relationship between the CPN and the DCU, this information can be

mapped into a SIM in the CPN. The predefined, same-structured SIM enables CPN to be standardized for production, copied, and manufactured in large numbers to function as "plug-and-play" nodes in buildings. Moreover, each CPN has six data ports and can be connected to up to six CPNs for data interaction without addressing. All the CPNs are connected to form a decentralized, flat-structured parallel computing network based on the spatial distribution of the building or the operation mechanism of the equipment. The various system calculation and control tasks in the building are completed through the data interaction of adjacent CPNs [6, 7].

3.2. Motivating Example. We analyze a case of a personnel evacuation application in I^{2}B and use it as an example to show how to solve I^{2}B application tasks and to present the challenges faced by developing I^{2}B APPs.

When a fire occurs in a building, people need to be evacuated immediately. In I^{2}B, based on the building information system, real-time information can be obtained regarding the distribution of the number of people in the building. According to this real-time information, the safest and quickest evacuation route can be dynamically planned, and the crowds can be guided to evacuate reasonably by the displays placed in each room and the main traffic avenues. The escape model introduced below uses a distributed recursive method to select the shortest path. During the evacuation process, only the CPNs mounted on the space units, such as rooms and corridors, participate in this calculation. According to the building topology network, these CPNs are automatically connected to the fire escape personnel evacuation function subnet, as shown in Figure 1.

When a fire occurs in any area, the corresponding CPN in that area first sends an alarm signal to notify all areas to enter the fire protection mode through diffusion transmission and then triggers this application. Assuming that the escape direction for a current space i points to space j, the escape time T_i is the sum of the crowd's moving time T_{ij}^{move} from space i to space j, the transit time T_{ij}^{pass} through space j, and the escape time T_j of space j [46], that is:

$$\begin{aligned} T_i &= \min_j \left\{ T_{ij}^{\text{move}} + T_{ij}^{\text{pass}} + T_j \right\}, \\ j &= 1 \sim 6, \\ T_{ij}^{\text{move}} &= \text{moveTime}\left(L_{ij}, \sigma_i, f_i\right), \\ T_{ij}^{\text{pass}} &= \text{passTime}\left(\sigma_i, \sigma_j, w_j\right), \end{aligned} \tag{1}$$

where the moving time T_{ij}^{move} is related to the average distance L_{ij} from the space i to the entrance and exit of space j, the fire severity f_i in the space, and the density σ_i of the personnel in space i and the passing time T_{ij}^{pass} is related to the density σ_i of the personnel in space i, the density σ_j of the personnel in space j, and the passage opening size w_j in space j. The escape time T_i changes in real time as the crowd flows, the fire develops, and the crowd crowding degree changes.

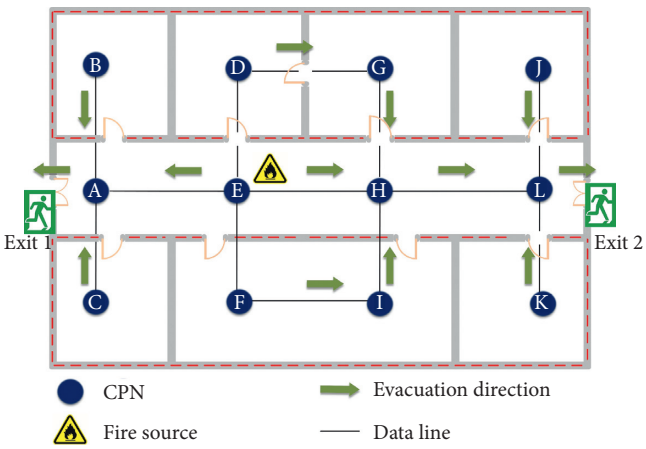


Figure 1: Personnel evacuation diagram.

When multiple adjacent spaces exist in space i, it is necessary to separately calculate the escape time for each adjacent space as the escape direction and adopt the space with the smallest escape time as the best escape direction to define the escape path. For example, space i has two neighbors j and k, and the escape time T_i is the minimum of T_{ij} and T_{ik}. The calculation process uses a recursive algorithm continuously. The CPN in each space unit continuously exchanges information with its neighboring CPNs to continuously establish a path from the security exit to the internal space. In the end, all the rooms will find the fastest escape path. The specific calculation steps are shown in Figure 2.

In this scenario, several difficulties occur when developing I^{2}B APPs:

(i) How to concisely describe the application tasks involved in personnel evacuation

(ii) How to set constraints to allow CPNs that meet the conditions to participate in the calculation

(iii) How to describe the interaction mechanism of CPN

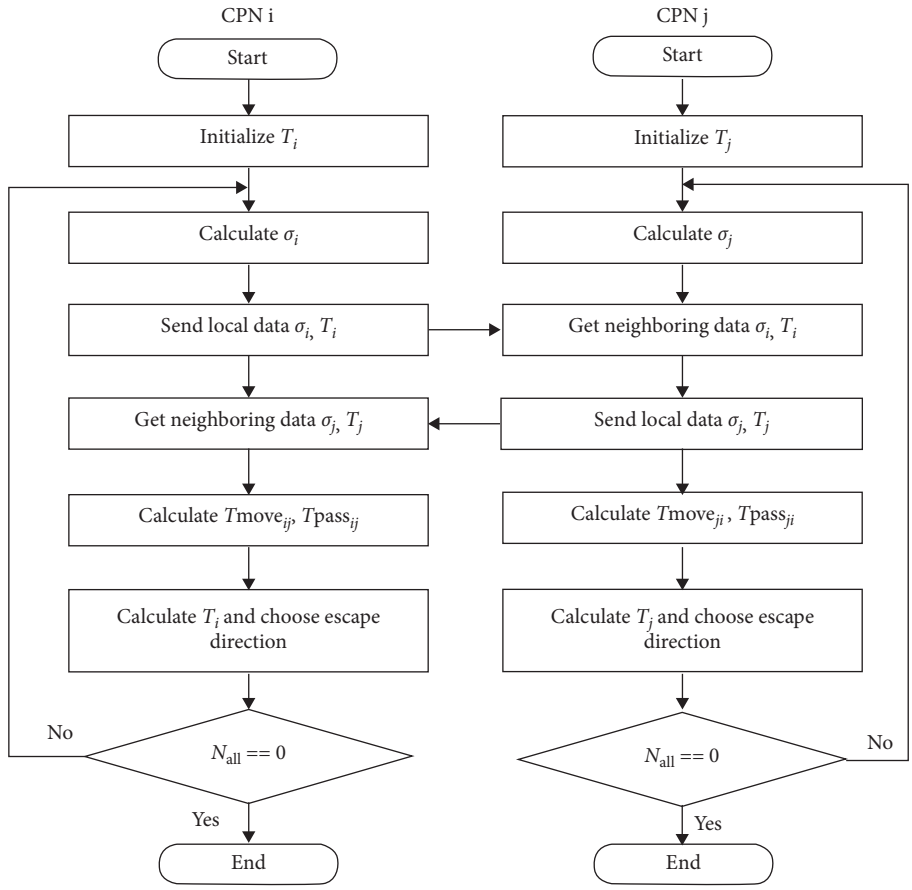
(iv) How to directly and quickly call the data in SIM during programming

(v) How to reuse the personnel evacuation application in buildings with different structures

3.3. Domain Analysis. Domain analysis is concerned with the analysis of the basic properties and requirements of the problem domain [47]. I^{2}B integrates human and machine, information and physics, and hardware and software and is a comprehensive and cross-cutting building intelligent platform. We divide the analysis of the I^{2}B domain characteristics into three aspects: human, buildings, and platform, as shown in Table 1.

Based on the analysis of domain characteristics of I^{2}B, the core requirements of the programming language used for I^{2}B APP development are summarized in Figure 3.

3.3.1. Simple and Friendly (R1). Due to the weak programming ability of most users and the difficulty in understanding and mastering the control logic of I^{2}B



T: Escape time; σ_i: Personnel density ; N_{all}: Number of people in all spaces;
Tmove$_{ij}$: Moving time from space i to j; Tpass$_{ij}$: Passing time from space i to j;

Figure 2: Calculation process in the personnel evacuation APP.

application tasks, a simple, friendly, and easy-to-use programming language with a high level of abstraction is needed, and complex details need to be encapsulated and hidden to reduce the burden of users. Furthermore, the programming language should have good integrability with existing development tools (e.g., Eclipse) to lower the user's threshold.

3.3.2. Focus on Building Element Characteristics (R2). The programming language should be oriented to building element characteristics and support an explicit description of electromechanical equipment information (operation state and operation mechanism), spatial unit information (environment state), the topology of the equipment pipe network, and spatial distribution so that developers can quickly describe building elements and easily maintain I^{2}B applications.

3.3.3. Focus on I^{2}B Platform Characteristics (R3). The programming language should be able to effectively describe I^{2}B platform characteristics, specific domain concepts (e.g., CPN, SIM, and decentralized CPN network), and distributed and parallel computing processes (e.g., adjacent CPN interaction and trigger mechanism) and improve the accuracy and efficiency of APP development.

4. Overall Framework for Touch Language Development

The Touch language development solution proposed in this paper follows the consistent DSL development method proposed in the literature. The iterative process proposed by Mernik et al. [10] includes four phases, namely, (i) decision (ii) analysis, (iii) design, and (iv) implementation. These phases constitute the formal definition of the DSL development process [47], as shown in Figure 4. In this section, each development phase of Touch is introduced in detail, and the DSL patterns used in each phase are also discussed.

4.1. Decision Phase. The goal of the decision phase is to determine the requirements for Touch language in the I^{2}B domain and to demonstrate that the investment in creating the Touch language can be compensated in the subsequent development and maintenance of the I^{2}B APP. The Touch language is designed to enable users with little I^{2}B domain knowledge and programming expertise and even end-users with I^{2}B domain knowledge but virtually no programming

Table 1: Domain analysis of I^2B.

Aspect	Domain feature	Description
Humans	Building engineers, building operation and maintenance managers, program developers	It is generally difficult for them to have both strong programming skills and domain knowledge about the I^2B system architecture and operating mechanisms.
Buildings	Diversity of building elements	The types of facilities and equipment in the building are complex, including rooms, corridors, and other spaces, as well as electromechanical equipment systems such as chillers and pumps.
	Spatial distribution characteristics	Spaces and equipment form space topologies and equipment pipe networks of different structures according to space distribution.
Platform	I^2B properties	As a new type of intelligent building, I^2B has its unique components and architecture, such as CPN, SIM, and decentralized CPN network.
	Network architecture	Each CPN is equal and there is no control center. Each CPN only connects and interacts with its neighbors to complete distributed and parallel computing.
	Consistency	The algorithms running on all CPNs participating in the same application task are identical.
	Real-time read and write	The building information stored in the SIM can be read and written in real time.
	Constraint	For a certain application task, not all CPNs in the CPN network can participate in the calculation, but only CPNs that meet the specific constraint conditions.
	Triggering	The calculation can be triggered by a certain CPN in the network.
	Universality	The scale and topology of the network formed by CPN are unknown, i.e., the application can automatically adapt to systems of various structures, but the system structure cannot be described specifically during programming.

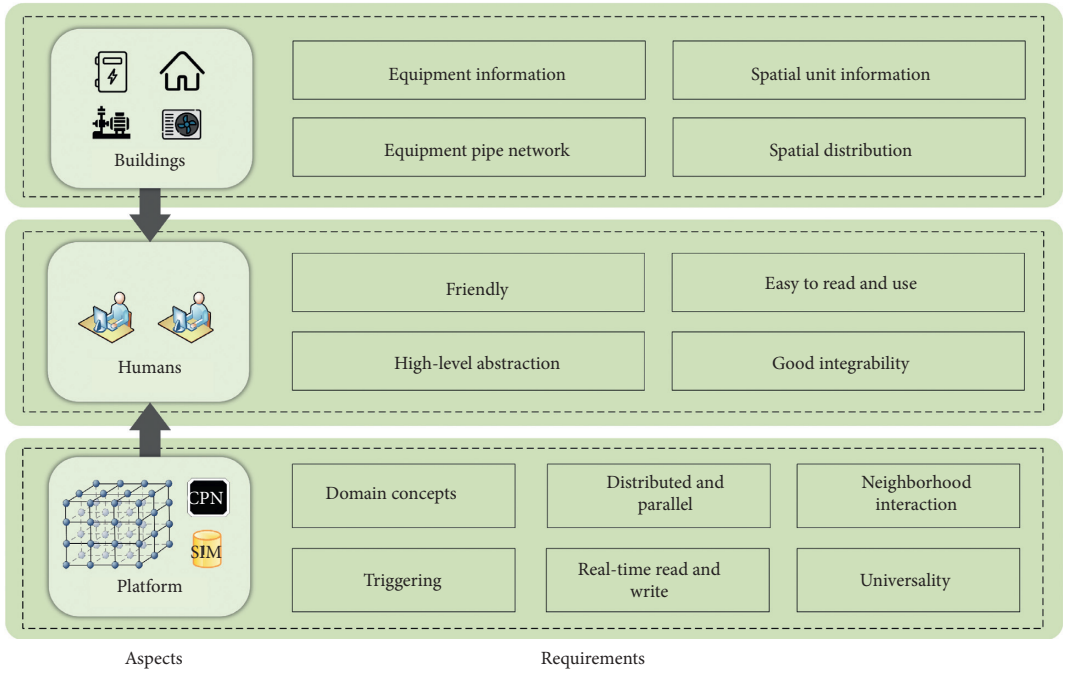


Figure 3: The core requirements of the programming language used for I^2B APP development.

expertise to develop I^2B APPs. To make a decision, stakeholders (including domain experts and language engineers) need to discuss the requirements of the I^2B domain by the DSL development patterns.

For the decision phase, the AVOPT (analysis, verification, optimization, parallelization, and transformation) [10] pattern is used in this paper to help the decision-making process. Analysis, verification, optimization, parallelization, and application transformation that are specific to the I^2B domain and written in GPL (e.g., C language) are usually not feasible because the source code patterns involved are too complex or not well defined, so using Touch language makes these operations possible.

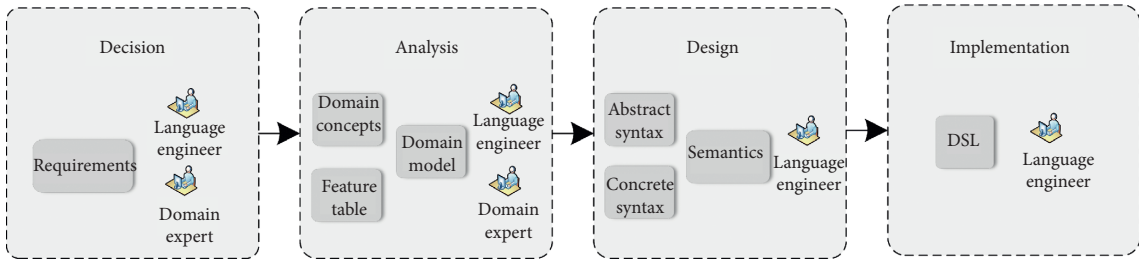


Figure 4: DSL development process and its phases (source: adapted from [47]).

4.2. Analysis Phase. At this phase, the domain characteristics of I^{2}B are analyzed. Domain experts help language engineers define I^{2}B domain concepts, feature descriptions, and functions and establish the core requirements of programming languages used for I^{2}B APP development.

For the analysis phase, an informal (the domain is analyzed in an informal way) [10] pattern is used. This is because this paper does not use formal domain analysis methods such as DARE (Domain Analysis and Reuse Environment) [48] and FAST (Family Oriented Abstractions, Specification, and Translation) [49] in this phase. Moreover, after the domain analysis, several terminologies specific to the I^{2}B domain are given instead of a formal domain model. Furthermore, I^{2}B domain knowledge in the process of analysis is derived from existing technical documents, knowledge provided by domain experts, and customer surveys, rather than existing GPL codes.

4.3. Design Phase. At this phase, the programming conceptual architecture of the Touch language is proposed, including three basic programming concepts: *Basic Unit*, *Neighborhood*, and *Domain*. The metamodels of the Touch language are also established using different abstract objects and UML notations of relations. Furthermore, we provide the key language elements and formal concrete syntax of Touch.

For the design phase, the DSL design method can be described along two orthogonal dimensions: (i) the relationship between DSL and existing languages and (ii) the formal nature of the design description. For the first dimension, the language exploitation pattern (DSL uses (part of) existing GPL or DSL) [10] is used. Because the Touch language is built based on existing C language, Touch piggybacks special language elements specific to the I^{2}B domain to part of the C language (e.g., common data types, if and while statements, etc.). For the second dimension, the formal [10] pattern is partially used because this paper specifies the syntax of Touch using formal symbols (e.g., regular expressions and grammars for syntax specification). However, this paper does not use attribute syntax, rewrite rules, and an abstract state machine to give the semantic specification of the Touch language, which will be our next work.

4.4. Implementation Phase. At this phase, Touch language support tools (i.e., I^{2}B APP development tools) are implemented, including the editor and code generator. This tool has the functions of keyword syntax highlighting, syntax error-checking, the automatic recommendation of standard information model parameters, and code generation and can effectively support Touch language program editing and the conversion of Touch to CPN-executable target code, improving the efficiency of I^{2}B APP development.

For the implementation phase, the compiler/application generator [10] pattern is used. Because this paper does not implement the Touch language through preprocessing, embedding, extending existing compilers, etc., but uses the Xtext tool (a widely used framework for DSL development) [50] to fully develop the editor and code generator of the Touch language, it can directly convert Touch programs into CPN-executable target code.

5. Programming Conceptual Architecture of Touch

Motivated by the example of personnel evacuation and the domain analysis of I^{2}B, the conceptual programming architecture of Touch is first designed. Our previous work [29] proposed INR, a programming model for I^{2}B APP development. INR contains three submodels, namely, *Individual*, *Neighborhood*, and *Region*, which are defined and implemented for describing different task requirements. However, after using the proposed INR programming model for two years, we find it still has the following limitations:

Limitation 1: the *Individual* submodel ignores the characteristics of some parameters in CPN SIM that change with time. Therefore, the INR programming model can operate only on the SIM parameters that exist at the current moment, which limits its ability to describe the I^{2}B application tasks.

Limitation 2: in the *Neighborhood* submodel, the "mailbox" type of adjacent CPN interaction method is too concerned with the complex message sending and receiving mechanism at the base of the node architecture, resulting in redundant, overly complex programs.

Limitation 3: in the *Region* submodel, the "originator" concept violates the principle that all CPNs in I^{2}B are equal. The explicit definition of "originator" results in poor application portability. Since the same application will be downloaded and run on all the CPN nodes, the same variables are defined in each CPN; thus, the concept of "regional variables" is redundant.

Limitation 4: the INR programming model is generally more complex and pays too much attention to specific computer calculation methods, resulting in low code readability and raising the learning cost for I^{2}B users.

In addition to the deficiencies in INR, this paper analyzes the physical entities that comprise the I^{2}B platform, improves the level of programming abstraction and the programming mechanism over that of the INR programming model, and provides Touch programs that are more readable and better suited to users. We propose a new programming conceptual architecture for I^{2}B APPs that includes three basic concepts to support I^{2}B APP development, i.e., the *Basic Unit*, *Neighborhood*, and *Domain*, as shown in Figure 5.

To improve readability and intuitiveness, we also establish metamodels for each of the three basic programming concepts using different abstract objects and UML notations of relations.

5.1. Basic Unit Programming Concept

5.1.1. Requirement Analysis. Each unit is managed by a CPN, which forms the "base" of the entire I^{2}B system. The SIM preset in a CPN contains various state parameters and physical property parameters of the unit. Due to functional differences, many types of units exist in I^{2}B, including building space, chillers, cooling towers, etc. An application must be executed in a matching type of unit. For example, the personnel evacuation application task requires only CPNs corresponding to building space units to participate in the calculation. A building space unit can control and solve local tasks through its built-in control logic. For example, in the personnel evacuation case, the CPNs can be used to solve the problem of how fast people are moving locally by using known parameters such as the local personnel density and the fire severity. This paper defines the minimum component unit for a CPN of I^{2}B, which is called a *basic unit*. The metamodel of the *basic unit* programming concept is shown in Figure 6, which uses programming abstractions, status attributes, functional attributes, and physical entities to describe different abstract objects and UML relations between them. Programming abstraction is used to describe the basic elements of control objects in application tasks. Status attributes and functional attributes are used to describe the structure of required data and functional effects of programming abstractions.

Definition 1. Basic Unit. The basic unit is the smallest unit participating in I^{2}B network calculations. Basic units are abstractions of the basic building units in the physical world. A basic unit can be defined as a quad:

$$\text{basic unit} = \left\langle B_{\text{name}}, B_{\text{type}}, B_{\text{data}}, B_{\text{auto}} \right\rangle, \tag{2}$$

where

(i) B_{name} represents a unique name that identifies the *basic unit* in the APP and must be defined manually.

(ii) B_{type} represents the type attribute of each *basic unit*, reflecting the functional characteristics of that particular *basic unit*.

(iii) B_{data} represents a collection of *basic unit* data members, corresponding to the SIM contained in the CPN. $B_{\text{data}} = D_s \cdot D_p$, where D_s represents the set of state parameters of the *basic unit*, including various operating state parameters such as the temperature and the density of people in the building space unit and D_s is a function of time, recorded in the form of time-series variables, $D_s(\tau) = [d_{s1}(\tau), d_{s2}(\tau), \ldots, d_{sk}(\tau)]^T$.
D_p represents a set of physical attribute parameters for a *basic unit*, such as the area of a building space unit and the power of a chiller, where $D_p = [dp_1, dp_2, \ldots, dp_k]$.

(iv) B_{auto} represents an autonomous function of the *basic unit*, and it is the entry point for APP execution inside the *basic unit*. B_{auto} is unique in an APP and contains a special type of dynamic behavior—events E, which corresponds each specific application task. $E = E_{\text{trig}} \cup E_{\text{non}}$, where E_{non} represents a set of nontriggering events and E_{trig} represents a set of triggering events. For example, the personnel evacuation case is a triggered event. When a building space unit detects a fire, the application task is triggered. Events must be created in autonomous functions, and a parallel relationship exists between events.

The *basic unit* programming concept accurately describes the time-varying characteristics of the information in the SIM using B_{data} and eliminates the "slave devices" concept bound to the tags and the redundant "static functions" concept of the *individual* programming model in INR [29], which can better meet the programming requirements of local tasks.

5.2. Neighborhood Programming Concept

5.2.1. Requirement Analysis. The *basic unit* programming concept can be used to describe local control tasks. However, global control of the I^{2}B building system is achieved through data interactions between the *basic units*. Each CPN in I^{2}B supports connections with up to 6 neighboring CPNs through data lines; thus, a *basic unit* communicates only with other *basic units* that are physically or logically adjacent without requiring addressing. The *basic units* that can communicate directly in this way are called neighbors. Through data interactions between neighboring *basic units*, a network parallel computing mode is formed. Based on this, we define a neighbor set of the *basic unit* in I^{2}B, called a *neighborhood*. The metamodel of the *neighborhood* programming concept is shown in Figure 7, where the identity attribute is used to describe the identifying features of programming abstractions. Moreover, the connotation abstraction means that it is not used in the description process, but the actual affected object.

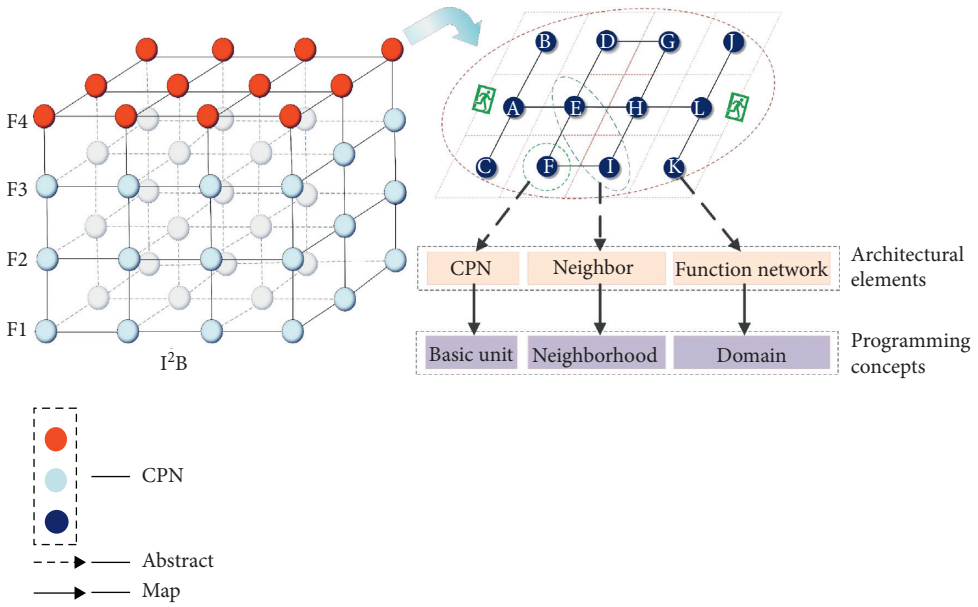


Figure 5: The programming conceptual architecture of an I²B APP.

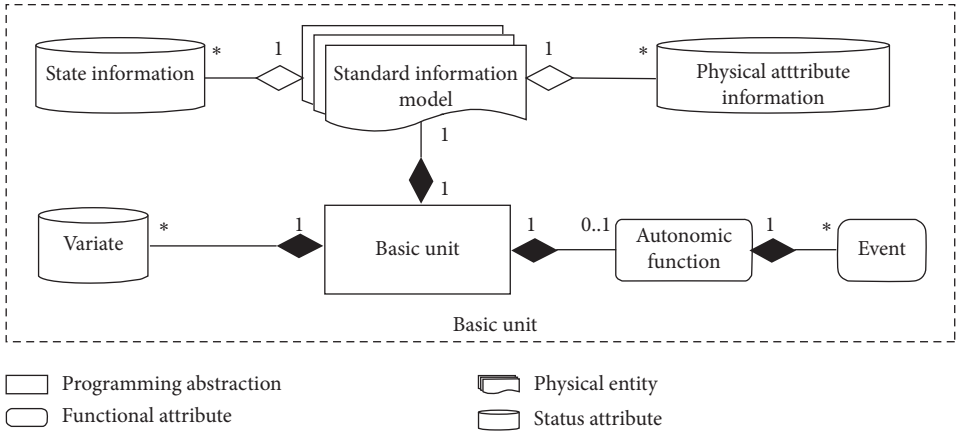


Figure 6: The metamodel of the basic unit programming concept.

Definition 2. Neighborhood. The neighborhood is a collection of nodes that are adjacent to the basic units participating in I²B network tasks. A neighborhood can be defined as a quad:

$$\text{neighborhood} = \langle \text{self}, \text{neighbors}, N_{\text{type}}, N_{\text{op}} \rangle, \quad (3)$$

where

(i) Self represents the abstraction of the *basic unit* in the *neighborhood* relationship.

(ii) Neighbors indicate the set of neighboring *basic units* that form a *neighborhood* relationship within the ontology.

(iii) N_{type} represents the type of functional network to which the *basic unit* in the *neighborhood* belongs. For example, the *neighborhood* of a building space unit X in I²B may include a water pump basic unit Y and a building space basic unit Z, where Y belongs to the water supply function network and X and Z belong to the personnel evacuation function network. In the personnel evacuation case, X interacts only with Z, so when data must be exchanged with a neighbor, the functional network type to which the neighbor belongs must be specified.

(iv) N_{op} represents a set of *neighborhood* operations, including various complex operations performed by a *basic unit* itself on the neighbor nodes, such as maximizing or summing certain data in the neighbor.

The proposed *neighborhood* programming concept in this paper uses a "direct" neighbor CPN interaction method, that is, the node can directly fetch the neighbor node's data without the need for displayed sending or receiving operations. Compared with the "mailbox" interaction method in INR [29], this method simplifies the message sending and receiving mechanism, shields the data dependency and

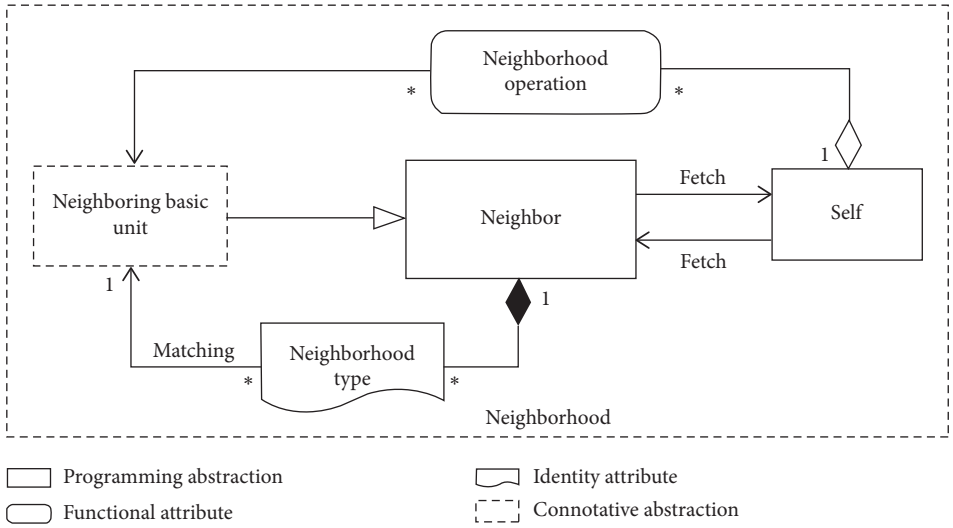


Figure 7: The metamodel of the neighborhood programming concept.

sequence dependency of variables, and reduces the size of the program. In addition, we propose the concept of "neighborhood type" to replace the "tag" concept in INR that requires a lot of cumbersome configuration due to its flexibility to identify the functional subnet type to which the CPN's neighbors belong and then select the corresponding neighbors for calculation.

5.3. Domain Programming Concept

5.3.1. Requirement Analysis. Based on the I^{2}B system network, various functional subnetworks can be defined based on the specific type of attributes and physical parameter information of each *basic unit* coupled with specific service function requirements. These subnetworks directly correspond to various types of physical networks and electromechanical systems in the physical space of the building, such as air conditioning system networks and personnel evacuation networks. Because the programming concept of a *neighborhood* is locally oriented, communication and control are limited to the *basic unit* and its neighboring nodes; thus, the network computing tasks within the functional subnetwork range cannot be described globally. This paper terms such a functional subnet of I^{2}B as a *domain*. The metamodel of the *domain* programming concept is shown in Figure 8.

Definition 3. Domain. A domain is a virtual network with a certain function formed by several basic units arranged in a certain topology. The nodes constituting a domain can include different types of basic units. A domain can be defined as a triple:

$$\text{domain} = \left\langle D_{\text{type}}, D_{\text{name}}, D_{\text{unit}} \right\rangle, \tag{4}$$

where

(i) D_{type} represents the type of *domain*, which corresponds to the type of functional subnets. D_{type} describes several types of *basic units* that the *domain* must contain, and the definition of D_{type} is clearly stated in the APP.

(ii) D_{name} represents the name identifier of the *domain*, defined by the APP developer according to D_{type}; one D_{type} can instantiate multiple D_{name}. After the types and locations of all the CPNs in the I^{2}B network have been determined, various types of *domains* are automatically generated such that the D_{name} of each *domain* is unique. In other words, a D_{name} corresponds to a specific *domain* in I^{2}B and is also unique within the same APP.

(iii) D_{unit} represents the set of specific *basic units* contained in the *domain*. After a *domain* is determined, the configuration personnel of the CPN network can establish the mapping relationship between a *domain* name and the *basic unit* members in that *domain*. $D_{\text{unit}} \supseteq \{B_{\text{name}}, B_{\text{num}}\}$, where B_{name} represents the *basic unit* name and B_{num} represents the *basic unit* number. When a *basic unit* in the *domain* reaches the trigger condition for an application task, it becomes the originator of that application task and triggers the application task to be executed on all *basic units* in the *domain*. Then, according to the principle of depth-first traversal, each *basic unit* in the *domain* is numbered from the origination point: the originator number is 0, and the remaining *basic unit* numbers are assigned numbers incrementally. There is a one-to-one mapping relationship between B_{name} and B_{num}.

The *domain* programming concept uses the "domain type" to describe different types of functional subnets and to show the constraints that participate in network computing, which is more intuitive than the "region constraints" concept of INR [29]. In addition, we propose D_{unit} representing the mapping relationship between the *basic unit* name and number to realize the precise positioning of a node in the *domain*. Furthermore, the concept of "originator" that is not conducive to application portability and redundant "regional variables" concept of INR are eliminated.

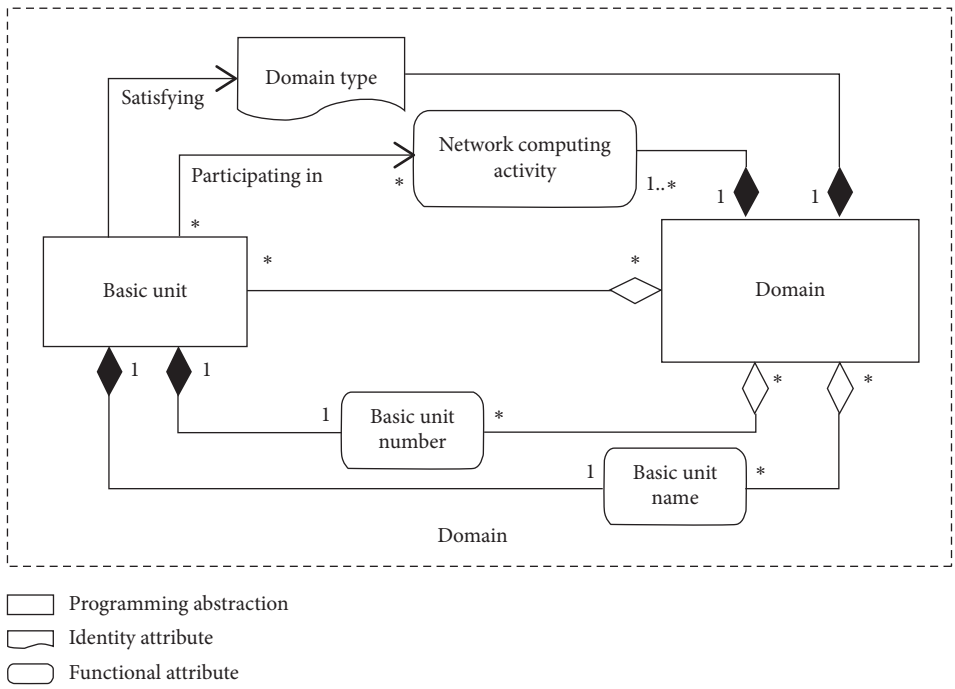


Figure 8: The metamodel of the domain programming concept.

6. Key Language Elements of Touch

Based on the programming conceptual architecture in Section 5, we provide the key language elements of Touch. An excerpt of the metamodel of Touch language elements is established, as shown in Figure 9. According to the service object, the key language elements are divided into three parts: one part serves the building domain (*basic unit*, *neighborhood*, and *domain*), one part is dedicated to CPN computing, and one is for controlling program. Furthermore, we use abstract objects and UML notation to describe the relationships between the key language elements. Each programming language element is formally defined.

6.1. Language Elements for the Building Domain. This paper proposes language elements for the building domain that can effectively describe the physical entities of I^{2}B and enhance the descriptive nature of Touch for I^{2}B application tasks. These language elements represent building domain elements in the form of syntax symbols that can be used directly in programming; they include both special data types and domain-specific variables, as shown in Table 2.

6.1.1. Special Data Type. When developing the I^{2}B APP, it is usually necessary to effectively describe the building elements in the I^{2}B, such as the attribute characteristics of the *basic unit* and the CPN functional network. However, due to the lack of language elements that directly reflect the characteristics of building elements, it is difficult for humans (e.g., building engineers and domain experts) in the I^{2}B domain to develop and maintain applications in a concise manner. To conveniently describe the type attributes of building elements, we propose special data types, which correspond to some of the programming concepts in Section 5, including the *class type*, *domain type*, and *neighborhood type*.

(1) Class Type. Because the types of *basic units* are quite diverse, such as the building spaces, boilers, and chillers, the functions of different types of *basic units* are also different. To identify the type of the *basic unit*, we propose the *class type* concept, which is mapped from the B_{type} described in Section 5.1 and shown in Table 2. The kind of *class type* is determined and corresponds to several basic units [6] specified in the I^{2}B system.

(2) Domain Type. Each type of functional subnet performs the same functions and contains the same *basic unit* types, but it instantiates multiple functional subnets of the same type according to the different spatial distributions and pipe network connections. When participating in an I^{2}B application task, the CPN must determine whether it belongs to the functional subnet type corresponding to the application task. Therefore, the type of functional subnet must be reflected in the programming language. We propose the *domain type* concept, which is mapped from D_{type} in Section 5.3 and includes such *domain types* as "domainTypeEscape," "domainTypeAHU," and "domainTypePump".

(3) Neighborhood Type. The neighboring nodes that are physically connected to the CPN with data lines may belong to different *domains*. To facilitate the unified operation of neighbor nodes belonging to the same *domain*, we propose a *neighborhood type*. The *neighborhood type* is mapped from the N_{type} in Section 5.2 and is used to describe the *domain type* to which the neighbor belongs.

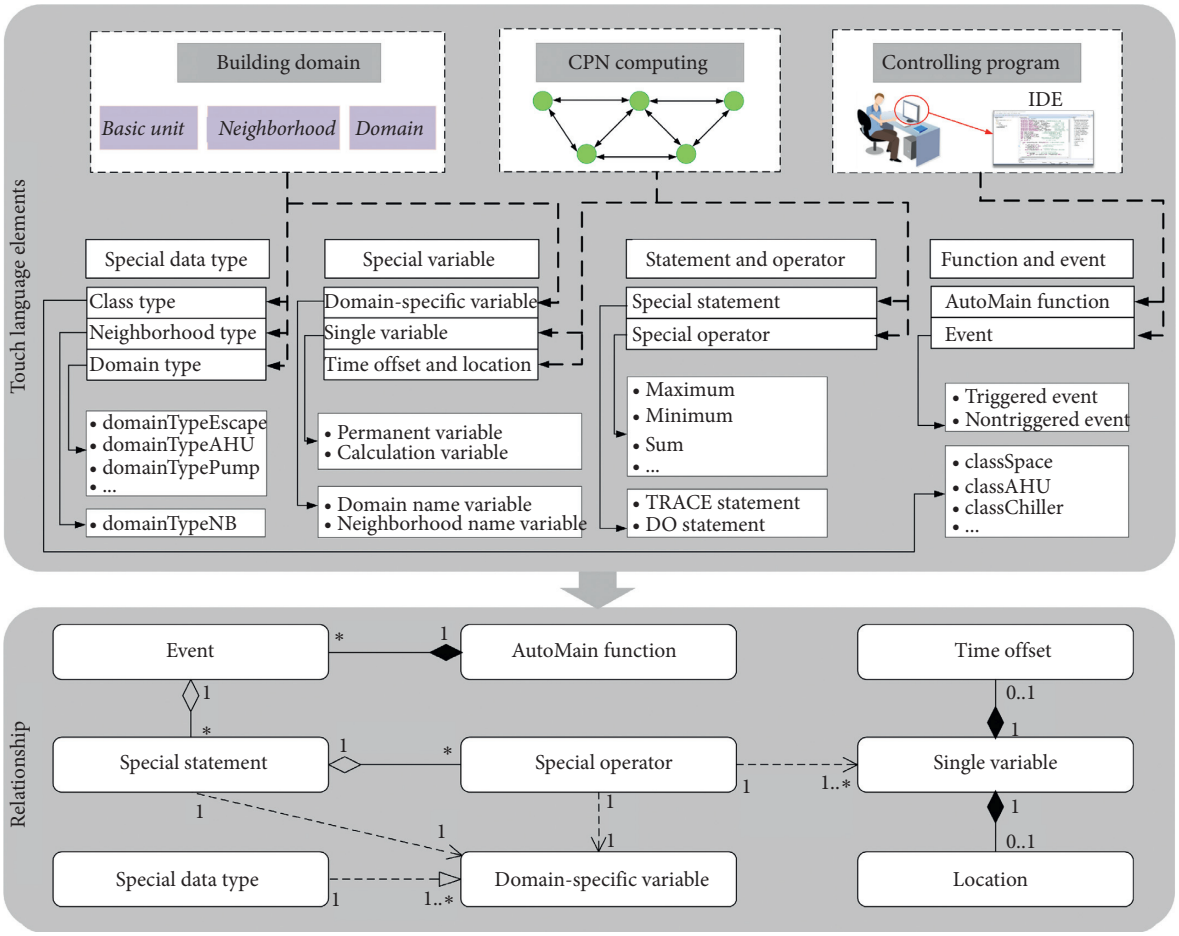


Figure 9: An excerpt of the metamodel of Touch language elements.

Table 2: Language elements for the building domain.

Special data type			Domain-specific variable
Class type	Domain type	Neighborhood type	
classSpace	domainTypeEscape	domainTypeNB	Domain name variable
classChiller	domainTypeAHU		Neighborhood name variable
—	—		

6.1.2. Domain-Specific Variable. There is an instantiation relationship between a type and a real object. To facilitate direct manipulation of objects, this paper proposes domain-specific variables that are divided into *domain name variables* and *neighborhood name variables*.

(1) Domain Name Variable. A *domain type* in I^{2}B may correspond to multiple *domains*. For example, there may be several *domains* with a type of "domainTypeEscape" in a building. While these *domains* have the same *domain type*, the specific CPN nodes they contain are different. Therefore, this paper proposes the *domain name variables*, which are mapped from the D_{name} in Section 5.3 and are variables instantiated from the *domain type* to characterize the *domain* name. Assigning different values to the *domain name variable* of the *domain type* creates different *domain type* instances. To distinguish *domain name variables* from other variables, Touch specifies that *domain name variables* must begin with "D_." In the example of the personnel evacuation application shown below, a *domain* whose *domain type* is an escape *domain* is defined with the *domain* name D_Floor1, signifying that the *domain* is located on the first floor.

```
domainTypeEscape D_Floor1;
```

(2) Neighborhood Name Variable. The physically connected neighbors of a *basic unit* may belong to different *domains*, and *basic units* that participate in network calculations interact with their neighboring nodes in the same *domain*; therefore, the *domains* of the neighboring nodes must be identified. The *neighborhood name variable* is a variable obtained by instantiating a *neighborhood type* and is used to characterize the *domain* of the neighbors involved in the calculation. The *neighborhood name variables* must begin

with "NB_" in Touch. The following example of the personnel evacuation application defines a *neighborhood* whose type is an escape domain, and the *neighborhood name* is NB_Floor1.

```
domainTypeNB NB_Floor1;
```

6.2. Language Elements for CPN Computing. I^{2}B application tasks are accomplished through data interactions between neighboring CPNs. A CPN must continuously read data from its neighboring nodes, perform operations using the data from those nodes, and perform subsequent operations based on the operational results. However, due to differences in the number of neighbors during an interaction, it is difficult for an application design to eliminate dependence on the node number or the communication address, which reduces portability and complicates the underlying communication mechanism of the CPN, making it difficult to describe interaction modes concisely. To this end, this paper proposes language elements that represent CPN computing, including those for single variables, time offsets, locations, special statements, and special operators, as shown in Table 3.

6.2.1. Single Variable. I^{2}B effectively integrates the physical information in the building with the CPN computing platform, making part of the information have time attributes, which is very different from ordinary variables. To effectively support the CPN calculation and processing process, we propose a single variable, which is the basic object of I^{2}B APP calculation processing and includes *permanent variables* and *calculation variables*.

(1) Permanent Variable. In the process of CPN interactive calculation, the physical information parameters in the building, such as room temperature and the power consumption of the chiller, are calculated to achieve control of the building. Moreover, the types of physical information parameters in the building are very complicated and can reach thousands. In addition, these parameters involve cumbersome parameter data types, byte lengths, and other attributes, so the workload of variable definition for these parameters in I^{2}B applications is massive. The SIM in I^{2}B has classified all information parameters, specifying that the types of information parameters contained in the same type of *basic unit* are the same. Since SIM parameters are necessary to perform calculations in the application and to reduce the difficulty of using those parameters, from the perspective of APP development, we propose a *permanent variable*, which corresponds to the B_{data} values in Section 5.1 and is a variable used to characterize state and physical attribute parameters in the SIM.

During I^{2}B APP development, when the programmer enters a *class type*, the system automatically recommends a parameter list corresponding to the *basic unit* of the *class type*. The user selects the appropriate *parameter name* from the recommended list to define the corresponding *permanent variable*. This implements read-write access to various parameters in the SIM, saves complicated system configuration and information definitions, and achieves a connection between the application software and building entities. Touch specifies that *permanent variable*s must begin with "P_." In the following example of the personnel evacuation application, a *permanent variable* is defined corresponding to the density of people in the *basic unit* of building space.

```
classSpace.PersonnelDensity, D_Floor1 P_Density;
```

(2) Calculation Variable. When developing an APP, some variables must be defined to store data and calculation results. However, because the *permanent variable* has fixed physical properties, it can only be used for calculation in the calculation process and cannot be directly used as the load of the calculation result. We propose the concept of a calculation variable, which is an intermediate variable defined in the APP whose stored value is valid during the current life cycle of the APP. Moreover, the calculation variables must start with "C_." In the following code, a *calculation variable* is defined that corresponds to the movement times of people in the building space of a specific *basic unit*.

```
float,APP,D_Floor1 C_MoveTime;
```

6.2.2. Time Offset and Location. To characterize the time and spatial attributes of single variables, *time offsets* and *locations* are proposed.

(1) Time Offset. Since some parameter values in SIM are a function of time, their corresponding *permanent variables* consist of a set of sequenced data. The *permanent variables* involved in the calculation may be at different moments; to simply represent their time-series characteristics, the concept of a *time offset* is proposed. A *time offset* corresponds to τ in $D_s(\tau)$ in Section 5.1, which is the time attribute of a *permanent variable* and appears in the form of a suffix of the *permanent variable* in the program. The *time offset* allows users to accurately describe the value of *permanent variables* at different times, enhances the descriptiveness of the Touch language for I^{2}B application tasks, and enables the close integration of building information with I^{2}B application development. The following example of the personnel evacuation application represents judging whether the value of P_FireJudge before 5 seconds is equal to 0, where the first 0 represents that this *permanent variable* belongs to the local CPN.

```
while (P_FireJudge(−5, 0) ! = 0)
```

(2) Location. In I^{2}B, each *basic unit* in the same *domain* runs the same APP; thus, each *basic unit* includes the same variables. Touch proposes a "direct" communication mechanism in which data sent and received by the node are not required in the syntax, but data of the target node identified by location can be requested directly. To achieve this "direct" communication mechanism, we propose the concept of *location*, which is the spatial attribute of a single

Table 3: Language elements for CPN computing.

Single variable	Time offset and location	Special statement	Special operator
Permanent variable	Time offset	DO statement	Domain operator
Calculation variable	Location	TRACE statement	Neighborhood operator

variable. *Location* is oriented to neighborhood interactions and is the distinguished name of the node in the *neighborhood*. It takes a value from "*0*, *U*, *D*, *L*, *R*, *F*, *B*," which represent the *basic unit* itself and the *basic unit's* upper, lower, left, right, front, and back physically connected neighbors, respectively. The *location* appears in the form of suffixes of single variables in the program. Due to the *time offset*, *permanent variables* are not referenced in the same way as *calculation variables*; instead, *permanent variables* are referenced as "<*permanent variable*>'(' <*time offset*> ',' <*location*> ')'," while *calculation variables* are referenced as "<*calculation variable*> '(' <*location*> ')'." The following code represents that the value of the local *calculation variable* C_MoveTime is equal to the value of C_MoveTime of its right neighbor.

```
C_MoveTime(0) = C_MoveTime(R);
```

6.2.3. Special Statement. During the process of solving I^{2}B application tasks, the *basic unit* usually traverses each neighbor node in the same *domain* as itself and calculates relevant values based on each neighbor node. For example, in the personnel evacuation application task, the *basic unit* calculates the time required to move from itself to its neighbors. Loops are commonly used in general programming languages to implement traversal. However, the differences in the number of neighbors attached to different nodes make it impossible to determine the upper limit of the incremental variable in the *for statement*, making it difficult to build generalizable applications. The traversal process of the *for statement* is a serial operation over several nodes semantically; however, that makes it difficult to reflect the data synchronization and parallelism characteristics of the I^{2}B system.

To solve the above problem, this paper proposes some special statements, including a *DO statement* and a *TRACE statement*, which act on different objects. The *DO statement* implements parallel operations on *basic units* of the same type in the same *domain*. The *TRACE statement* traverses all the nodes in the *neighborhood* by specifying the *neighborhood name*, enabling the implementation of parallel operations on *basic units* in the *neighborhood*. The following example of the personnel evacuation application traverses the nodes in the *neighborhood* where the *neighborhood name* is NB_Escape. Dividing the distance from the local node to the neighbor node by the speed is equal to the time required to move from the local node to the neighbor. Using this statement, the local node can quickly find the moving time from the local node to each of its neighbors. Such operations reduced the amount of program code that must be written and exhibited universality.

```
TRACE node (NB_Escape)
    C_MoveTime (0) = P_Length(0,0)/C_MoveSpeed(0);
```

6.2.4. Special Operator. The *basic units* in I^{2}B application tasks usually need to perform complex operations on node data in a *domain* or *neighborhood*, such as maximizing or summing a variable in the *neighborhood*. This leads users to design specific algorithms to solve problems when programming. Since such algorithms must consider the number of *basic units* involved in the calculation and implement a complex underlying interaction mechanism, users consequently have to focus too much on the algorithm level instead of the application level. In addition, the code bindings to *basic unit* numbers cause difficulties when trying to achieve portability.

To improve the friendliness of the language and reduce the programming difficulty, this paper proposes special operators for network computing. According to the range of *basic units* that the operator acts on, the special operators are divided into *domain operators* and *neighborhood operators*. The syntax of a *neighborhood operator* is "<*neighborhood name*> '.' <*single-variable expression*> '.'<*special operator type*>," where *neighborhood name* specifies the scope of the operator, the *single-variable expression* represents the calculation object, and the *special operator type* includes *Maximum*, *Minimum*, *Sum*, and so on. Note that when referring to a *single variable* here, there is no need to add a *location* suffix because the operation is a unified operation on nodes in the *neighborhood*. The *neighborhood operator* replaces the node number with the *neighbor name variables*, which makes the program design effectively get rid of the dependency on the node number and become more flexible.

The following example specifies that in the personnel evacuation application, the minimum value of the sum of C_EscapeTime, C_PassTime, and C_MoveTime of the nodes in the *neighborhood* named NB_Escape is calculated, and the minimum value is assigned to the variable *t*. When special operators are used for network calculations, the complex algorithmic details and low-level interactions among nodes are shielded from both the user and the code, which improves program portability.

```
t = NB_Escape. [C_EscapeTime + C_PassTime +
C_MoveTime].Minimum;
```

6.3. Language Elements for Controlling Program. When a user writes an I^{2}B APP in a GPL, the lack of corresponding language elements constricts the APP, resulting in a lack of modularity and poor program readability. To provide the entry point for the APP and improve the encapsulation of the Touch program, we propose *AutoMain function* and *events* for controlling program.

6.3.1. AutoMain Function. The *AutoMain function* is the entry point and main function of the Touch language program and is mapped from the B_{auto} programming concept in Section 5.1. I^{2}B applications execute by starting at the beginning of the *AutoMain function* and continuing until the *AutoMain function* ends.

6.3.2. Event. A *basic unit* in I^{2}B may perform several different application tasks, some of which are characterized by event triggering. When using a GPL for programming, because the code organizational structure of the application program is not tightly constrained, it is often nonmodular, and programs involving different functional application tasks are often tightly coupled. This also makes it more difficult to achieve code reuse and can affect the reliability and maintainability of the program. To effectively describe the event triggering characteristics of application tasks and to enhance the modularity of application task programs, we propose *events*, which correspond to specific application tasks in I^{2}B. *Events* are mapped from the *E* programming concept in Section 5.1 and are divided into *triggered events* and *nontriggered events* according to the operation mechanism of application tasks. *Triggered events* are not triggered immediately after they are created in the *AutoMain function*; instead, they are launched when a triggering condition is met. The following example of the personnel evacuation application indicates that the trigger condition is reached when the *permanent variable* P_Fire (0,0) that corresponds to the fire judgment parameter is greater than 0, at which point the application task is executed and the local fire alarm is then turned on.

event PersonnelEvacuation(P_Fire(0, 0) > 0)
{P_AlarmSetValue(0, 0) = 1;}

Events are the key to parallel and triggered programming. They are the smallest unit of program execution flow in Touch, and different *events* can be executed in parallel. *Events* can effectively describe the event triggering characteristics of application tasks and encapsulate a program describing an application task to achieve program modularization.

6.4. Touch Concrete Syntax. Based on the conceptual programming architecture and key language elements, we provide an excerpt of the Touch syntax to show the most relevant features, as shown in Figure 10. The Touch program consists of a set of variable definitions and a main program. The variable definition statements can include both domain-specific variable definitions and single variable definitions. The main program contains a series of *events*. *SingleVarRef* represents the syntax when referencing a single variable, which is different from the definition of a single variable, and it has a suffix of *time offset* or *location*. Moreover, *stmt* represents various statements and *exp* represents expressions. *LibraryFunction* in the syntax refers to some library functions included in Touch, e.g., *Record* is used to record the best escape path. Due to space limitations, we did not show the syntax of the terminators corresponding to language elements such as *DomainVar*, *PermanentVar*, and *TimeOffset*.

7. Touch Language Support Tool Implementation

Because the Touch language has many special language elements oriented to the characteristics of I^{2}B domain and supports real-time read and write interaction with the SIM in CPN, it is quite different from the existing programming languages. Therefore, it is difficult for existing application development tools to directly support Touch language program editing and implement functions such as keyword syntax highlighting and syntax error-checking. In addition, the Touch language cannot be directly executed on the CPN, it first needs to be converted into a hexadecimal target code form that the CPN can recognize. Therefore, tools to effectively support the development of I^{2}B APP are still lacking. To improve the efficiency of users developing I^{2}B APPs, this paper builds IBADT (I^{2}B APP Development Tool) to automate the development process of I^{2}B APPs. The novelty of this tool lies in two aspects:

(i) IBADT provides a friendly GUI (graphical user interface) for editing and testing I^{2}B applications in an understandable Touch language.

(ii) IBADT can automatically convert I^{2}B applications based on Touch into hexadecimal CPN-executable code and insert it directly into the target software system, which effectively reduces the development burden of users who lack I^{2}B application task management control knowledge.

7.1. Functional Requirements of IBADT. IBADT aims to promote I^{2}B APP development and bridge the gap between I^{2}B application control and software development for users. Our functional requirements for IBADT are as follows:

RQ1: *Effectiveness*. The tool should effectively support the I^{2}B APP development process and should be able to edit and test I^{2}B application task control strategies based on Touch.

RQ2: *Efficiency*. The tool should have certain automation capabilities, such as automatic keyword syntax highlighting and syntax error-checking in Touch programs, and should be able to recommend various SIM parameters based on the *class type* entered and generate I^{2}B APP code files based on executable object code.

RQ3: *Availability*. The tool should be friendly and easy for I^{2}B APP developers (users) to use, and it should provide an intuitive user interface.

RQ4: *Reusability*. The tool should store I^{2}B applications in standard and open file formats to foster sharing and reuse.

```
Prog            ::=  VarDef MainProg
VarDef          ::=  DomainSpecVarDef | SingleVarDef
DomainSpecVarDef ::=  DomainType DomainVar | NeighborhoodType NeighborhoodVar
SingleVarDef    ::=  ClassType "." ParameterName "," DomainVar PermanentVar
                 |   DataType "," APP "," DomainVar CalculationVar
MainProg        ::=  DataType AutoMain { Event }
Event           ::=  event EventName TriggerCondition {stmt} | stmt
stmt            ::=  TRACE node NeighborhoodVar {stmt} | LibraryFunction exp | {exp}
TriggerCondition ::=  exp | Timer "(" on "," digit ")"
exp             ::=  SingleVarRef | SpecialOpExp | exp Operator exp
SpecialOpExp    ::=  [DomainVar | NeighborhoodVar] "." "["exp "]" "." OperatorType
OperatorType    ::=  Maximum | Minimum | Sum | Or | And | ...
LibraryFunction ::=  Record | Gather | Broadcast | …
SingleVarRef    ::=  PermanentVar "(" TimeOffset "," Location ")"
                 |   CalcutionVar "(" Location ")"
DataType        ::=  int | float| bool | NULL
ClassType       ::=  classSpace | classChiller | classBoiler | classElevator | ...
DomainType      ::=  domainTypeEscape | domainTypePump | domainTypeAHU | ...
NeighborhoodType ::=  domainTypeNB
```

Figure 10: An excerpt of the Touch syntax.

7.2. Key Approach to IBADT Development. To achieve these specific functions, a Touch-to-Hex-based IBADT development method is proposed. Figure 11 shows the general concepts behind this method. Combining the Touch language and the CPN-executable target code specification, we implement the I^2B APP development tool (IBADT) in Xtext [50], a widely used framework for the development of DSLs. When using IBADT to develop an I^2B APP, we can first edit and test the I^2B application using Touch language rules and then automatically convert the I^2B application from the Touch language into hexadecimal CPN-executable object code.

IBADT consists of three key parts: the editing and testing module, the persistent storage module, and the code generation module. Figure 12 shows a high-level view of IBADT architecture.

7.2.1. Editing and Testing Module. The editing and testing module is a basic component in IBADT that mainly supports editing and checking I^2B applications. First, Touch syntax rules are listed in the IBADT rule-editing area, and IBADT managers can modify these rules based on the actual needs of I^2B application tasks. When developing an I^2B APP, this module performs keyword highlighting and syntax-checking to ensure that the application meets the Touch standards. If a syntax error exists in the program, the incorrect code block is indicated with a red underline.

This module supports functionality such that when a *class type* of a *basic unit* is entered in the program editing area, all the parameters of the *class type* can be automatically recommended in the form of a member list from which users can select and apply. By defining and calling *permanent variables*, interoperation with SIM parameters is realized. Essentially, this feature points to the *permanent variable* memory in the SIM parameter table.

7.2.2. Persistent Storage Module. To achieve reading real-time parameter data from the SIM, a persistent storage module is designed. The persistent storage module provides storage and sharing services for SIM data. Because the SIM information has the characteristics of a tabular data structure, we use the standard MySQL database as a storage carrier and design a MySQL file structure for storing SIM data. This module inserts the *class type* of all *basic units* in the SIM and all the parameter names corresponding to each *class type* into the MySQL data table. The storage structure of the MySQL file is divided into two areas to store the SIM data: the SimClassType area and the SimParameter area. The SimClassType area is used to store the *class type* names in the SIM, and the SimParameter area is used to store the specific parameters and their various attributes corresponding to each *class type*, such as a parameter's name, attribute value, data type, and so on.

7.2.3. Code Generation Module. The ultimate goal of IBADT is to generate hexadecimal CPN-executable code and insert the I^2B application task logic directly into the target software to build the I^2B software system. To achieve this goal, we implement a code generator with Xtend [51], which is a Java-like general-purpose programming language tightly integrated with Java that allows writing much simpler and much cleaner programs. This generator parses each element in the I^2B APP and creates the corresponding CPN-executable code. Furthermore, this generator is integrated in the Xtext-based IDE (integrated development environment) of the Touch language.

7.3. Functional Description of IBADT. The main interface of the IBADT tool is shown in Figure 13. The program shown in the figure is the personnel evacuation application example of this paper. The left side of the main interface shows the project file directory. Users can view existing project files as

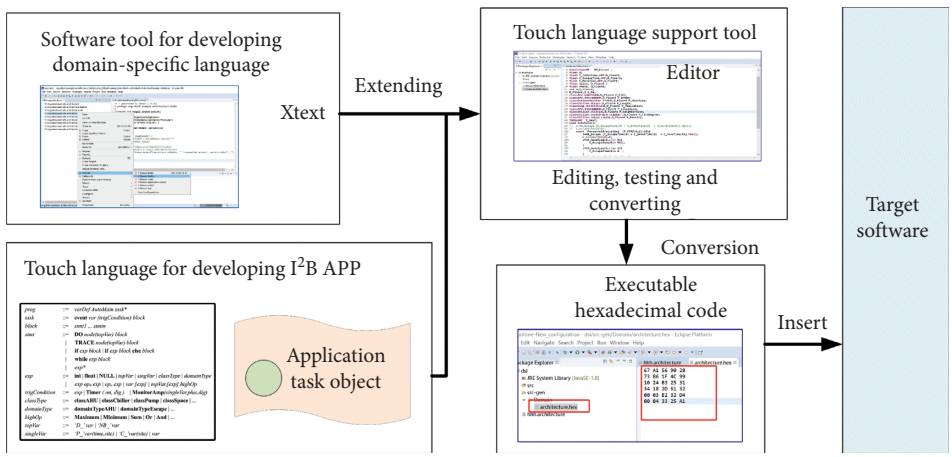


Figure 11: The general concepts behind the IBADT implementation.

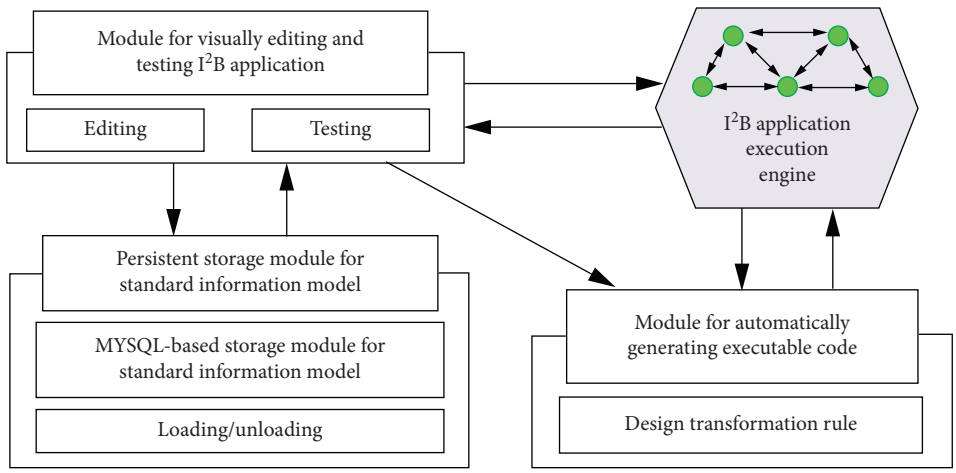


Figure 12: IBADT architecture.

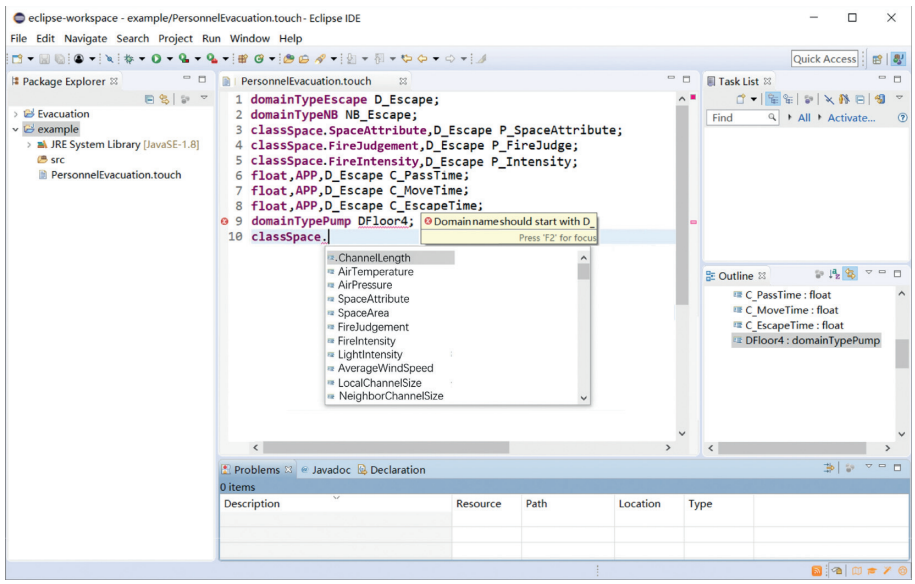


Figure 13: IBADT's main function display.

well as the object code files generated by the code conversion module. The right side of the main interface shows the newly created variable in the program and the data type corresponding to that variable. IBADT has the following functions.

7.3.1. Program Editing. The program in Figure 13 defines some *permanent variables*, such as P_FireJudge and P_Intensity, which represent the fire judge parameter and the fire intensity parameter, respectively. IBADT supports keyword highlighting in the Touch language, including data types and commands such as "domainTypeEscape," "classSpace," and "float". In addition, we can check the syntax of the Touch program at any point during the editing process. As shown in Figure 13, due to carelessness, we wrote "DFloor4" instead of "D_Floor4," which does not follow the rule that a *domain type* must start with "D_." However, IBADT intelligently found the problem segment "DFloor4," inserted underscores to indicate the problematic code segment, and displayed a "Domain name should start with D_" prompt.

7.3.2. Parameter Recommendation. Figure 13 shows that when the programmer enters "classSpace," IBADT automatically displays a SIM parameter list corresponding to the "classSpace." The parameter list contains "ChannelLength," "AirTemperature," and other values from which the user can select to rapidly define *permanent variables*.

7.3.3. Code Generation. Clicking the *Project- > Build Automatically* generates the executable object code file automatically on the CPN. The source file is the Touch program code file (∗ .touch) written by the user using the Touch language, and the automatically generated file is the object code file (∗ .hex).

8. Experimental Evaluation

In this section, the proposed method is evaluated to verify the effectiveness and the efficiency of Touch compared to traditional development methods. Effectiveness is examined from the applicability, portability, and maintainability of the application. Efficiency means that users must be able to develop I^2B APPs faster when using Touch than when using traditional development methods.

8.1. Experiment of Effectiveness. The personnel evacuation case (described in Section 3.2) is used to verify the effectiveness of Touch and its supporting tools. The experiment is designed to answer these questions:

(i) Can Touch be used in practical scenarios?

(ii) Can applications developed with Touch adapt to different network topologies?

(iii) Is the application developed using Touch easy to maintain?

We first develop a personnel evacuation application using Touch. Then, in the actual experimental scenario, the application is tested and compared with the execution effect of an application developed in C language.

8.1.1. Experimental Scenario. In this experiment, we select the first floor of our research group's experimental building as the experimental scenario. The first floor contains two emergency escape exits to the outside; these exits are located at both ends of the corridor and are termed Exit 1 and Exit 2. A floor plan view is shown in Figure 1. According to the building space unit division and passage connection rules, the building is partitioned into 12 building zones. Each zone is equipped with basic data acquisition and processing equipment such as a CPN, sensors, universal controllers, and dynamic evacuation indicators. The smoke temperature composite sensor in the zone can monitor fire information in real time. Simultaneously, personnel monitoring equipment installed in the partition monitors the location distribution information of the personnel present in each partition [52]. The dynamic evacuation indicator can change the escape direction based on the conditions to effectively guide decisions of trapped people to follow the best evacuation direction and route. The CPNs on the first floor of the experimental building are wired to construct a network. Figure 1 shows a CPN connection network topology diagram corresponding to the building zones. The initial distribution of people in the building is shown in Table 4, and the total number of people on the first floor is 64.

8.1.2. Development with Touch. Figure 14 shows the Touch-based application of personnel evacuation. Lines 1 and 2 define a *domain name variable* and a *neighborhood name variable*. This application task belongs to the personnel evacuation function subnet; therefore, the *domain type* is domainTypeEscape. Lines 3 to 6 define a series of *permanent variables*. Due to a large number of variables, only a few are listed here. Lines 7 to 9 define *calculation variables* that represent the moving time, passing time, and escape time. Line 12 indicates that an *event* named PersonnelEvacuation is defined in the AutoMain function. When the fire parameter P_Fire (0,0) is greater than 0, the *event* is triggered, and then all nodes in the *domain* begin computing in parallel. Lines 13 to 16 indicate that the local escape time is initialized according to the attributes of the building space unit. When this node is an emergency exit node, the escape time is 0; however, when it is an internal node, the escape time is NULL, where NULL represents infinity. Lines 18 to 21 use the *TRACE statement* to traverse the neighborhood nodes to calculate the passing time and moving time from the local node to each neighbor node. Line 22 uses the neighborhood operator to find the minimum of the sum of the passing times and moving times of the local CPN to each neighbor and the escape time of the neighbor. The escape paths of the corresponding neighbor nodes are also recorded, and finally, the local escape time is updated. Through iteratively computing these values, the escape path is updated in each iteration until the iteration termination condition

Table 4: Distribution of personnel on the first floor of the experimental building.

Basic unit	A	B	C	D	E	F	G	H	I	J	K	L	Total
Number of people	2	6	5	10	1	7	9	3	11	6	3	1	64

```
1  domainTypeEscape D_Escape;
2  domainTypeNB NB_Escape;
3  classSpace.SpaceAttribute,D_Escape P_SpaceAttribute;
4  classSpace.FireJudgement,D_Escape P_FireJudge;
5  classSpace.FireIntensity,D_Escape P_Intensity;
6  …
7  float,APP, D_Escape C_PassTime;
8  float,APP, D_Escape C_MoveTime;
9  float,APP, D_Escape C_EscapeTime;
10 float, D_Escape t;
11 voidAutoMain () {
12 event PersonnelEvacuation (P_FireJudge (0,0) >0 ) {
13         if (P_SpaceAttribute (0,0) == 0)
14             C_EscapeTime (0) = NULL;
15         if (P_SpaceAttribute (0,0) == 1)
16             C_EscapeTime(0) = 0;
17         while (P_FireJudge(0,0)!= 0){
18             TRACE node (NB_Escape)
19                 C_MoveTime(0)=P_Length(0,0)/moveSpeed(
                       P_FireIntensity(0,0),P_Density(0,0));
20             TRACE node (NB_Escape)
21                 C_PassTime(0)=P_People(0,0)/passSpeed(
                       P_LocalSize(0,0),P_NeighborSize(0,0));
22                 t = NB_Escape.[C_EscapeTime+ C_PassTime +
                       C_MoveTime].Minimum;
23                 C_EscapeTime(0) = t;
24                 RecordC_EscapeTime(0);}}}
```

Figure 14: Personnel evacuation application.

of line 17 is finally met. When the number of people in all spaces is 0, the calculation ends, and the personnel evacuation task is complete.

8.1.3. Experimental Process and Results. Based on the experimental scenario, we conduct three groups of personnel evacuation experiments. In Group 1, no dynamic evacuation instructions are given, while for Group 2 and Group 3, we compile and download the personnel evacuation APPs written in the C language and Touch, respectively, to the CPN. In both programs, the information interactions between the nodes are used calculate the evacuation instructions. During the experiment, personnel are evacuated in strict accordance with the provided instructions. After the hardware platform is prepared, IBADT is used to convert the personnel evacuation APP written in Touch to CPN-executable hexadecimal object code (stored in Touch.hex). Simultaneously, the GNU Compiler Collection (GCC) is used to convert the personnel evacuation APP written in the C language to C.hex files. We use the serial debugging assistant to download the code in both Touch.hex and C.hex to the CPN. Then, we run the APP corresponding to the C.hex code in Group 2 and the APP corresponding to the Touch.hex code in Group 3.

As listed in Table 5, in Group 1 (without instructions), the number of people who chose Exit 1 and Exit 2 accounted for 24.1% and 75.9% of the total population, respectively, while in Group 2 (with instructions), the number of people who chose Exit 1 and Exit 2 accounted for 45.3% and 54.7% of the total population, respectively. In Group 3 (with instructions), the number of people choosing Exit 1 and Exit 2 accounted for 40.6% and 59.4% of the total population, respectively. The results of these three sets of experiments show that the evacuation instructions provided by the I^2B system effectively reduce the evacuation pressure on individual exits by balancing the number of personnel using each exit.

The total evacuation time for Group 1 without instructions is 140 seconds, while the evacuation times for Group 2 and Group 3 are 125 seconds and 136 seconds, respectively, indicating that providing evacuation instructions reduces the total evacuation time. Comparing the experimental results of Group 2 and Group 3, it can be seen that the total evacuation times of the two experiments and the number of evacuations at each exit are similar and that the total evacuation times are shorter than evacuation without instructions. This shows that the effect of the personnel evacuation APP written in Touch is quite similar to the effect of the evacuation APP written in C.

8.1.4. Discussion. To evaluate the effectiveness of Touch, we now discuss its features with respect to the requirements of the language.

(i) *Applicability*. From the above analysis results, it can be seen that the I^2B personnel evacuation APP developed using Touch language can effectively balance the number of evacuees at different exits, avoid congestion, reduce the evacuation time, and ensure the effective evacuation of personnel in emergencies. This proves that Touch can be effectively applied to I^2B APP development.

(ii) *Portability*. Generally, people develop applications for a specific network topology in a building. When a new building is given, the application needs to be programmed from scratch. Touch unbinds the application from the network topology through language elements such as *DO statements* and *special operators*, making the program portable. We can abstract a general solution through Touch, which is applicable to any given building topology. If the number of CPNs in the personnel evacuation function network changes, the application in Figure 14 can still be applied without changing the code.

(iii) *Maintainability*. Touch enhances abstraction by introducing domain-specific concepts (e.g., *basic units*, *neighborhoods*, and *domains*) in the source code. This helps users to program in the same way as the case description and facilitates the writing, understanding, and maintenance of the applications, whereas using a general-purpose programming language such as C language is more difficult.

Table 5: Comparison of the experimental results.

Group	Is there an evacuation instruction?	Number of people evacuated at Exit 1	Evacuation time at Exit 1 (s)	Number of people evacuated at Exit 2	Evacuation time at Exit 2 (s)
1	No	13	32	51	140
2	Yes	29	94	35	125
3	Yes	26	102	38	136

8.2. Experiment of Efficiency. The purpose of introducing Touch in I^{2}B APP development is to reduce programming difficulty and improve development efficiency. This means that users can develop I^{2}B APPs faster with Touch and the understandability and learnability of Touch are better.

8.2.1. Experiment Planning and Design. In this section, we discuss the planning and design of experiments, including experimental subjects, experimental variables, experimental instruments, and experimental design.

(1) Subjects. The subjects in this experiment include 15 graduate students in computer science. All subjects have basic experience in application development using C and Visual Studio IDE. They have designed and developed software in an academic environment for at least one year. Most of them have a basic knowledge of I^{2}B and building control. The subjects are explicitly informed that they are free to choose whether to participate in the experiment and that their choice does not affect their course performance.

(2) Variables. We select the dependent and independent variables as follows. The dependent variable is the analyzed variable. In our experiments, the application development time, the number of lines of program code, and the development difficulty are the dependent variables. The development time and the number of lines of program code can be directly measured during the experiment while the development difficulty cannot be directly measured. The subjects can score the development difficulty of different development methods according to indicators such as ease of use, ease of learning, and so on. The independent variables are variables that we can control and change. In this experiment, we choose C language, the INR programming model [29], and Touch language as independent variables.

(3) Instrumentation. To support the experiments, we used the following materials:

(i) *Hardware Platform and Software Platform.* The experiments are performed in a laboratory (controlled environment). The computers in the laboratory have the same configuration and are uniformly installed with Visual Studio (version 15.0), Touch's supporting tool IBADT, and INR's supporting tool. The computer desktop also includes a teaching platform that supports homework submission, completion time recording, and filling in scorecards.

(ii) *Log Sheet.* We design two log sheets to collect experimental data. The first log sheet is used to record the application development time and the number of lines of code for each subject during the experiment. The second log sheet is designed as a scorecard, which covers a series of indicators that can reflect the difficulty of development, such as the understandability of the language, the friendliness of the development environment, and so on, as shown in Table 6. The subjects are asked to score the development method according to these concrete criteria.

(4) Experimental Design. Table 7 shows the design of our experiment. According to the principles of randomization, blocking, and balancing, the subjects are divided into three groups labeled as Group 1, Group 2, and Group 3, with 5 subjects in each group. During the preparation phase of the experiment, all participants are uniformly trained to understand the three application development methods. The experimental process is divided into two phases. In the first phase, the subjects use different development methods to complete the tasks. In the second phase, the subjects are asked to mark the development methods they used after completing tasks and submitting source code.

8.2.2. Experiment Operation. The operation procedure of the experiment is introduced in this section, including experiment execution, data collection, and data verification.

(1) Experiment Execution. Before the start of the experiment, we use three hours to conduct unified training for the subjects, including the following content: (i) the basic concepts of I^{2}B and its application development requirements (Section 3) are explained in detail; (ii) C language and its supporting tool Visual Studio are briefly reviewed; (iii) the INR programming model and its supporting tool are specifically introduced; and (iv) the Touch language and its supporting tool (Sections 5, 6, and 7) are specifically introduced.

The subjects are then divided into three groups, each group containing five subjects. The actual experiment is divided into two phases. In the first phase, subjects in Group 1 use C and Visual Studio to design and develop the personnel evacuation applications (Section 3.2) while subjects in Group 2 and Group 3 use INR and Touch to complete the same task. All subjects are required to complete the task independently, and they also need to complete the task quickly while ensuring correctness. During the execution of the experiment, the subjects can ask the experiment administrator (the second author of this paper) questions about language tutorials or task descriptions, but the administrator does not answer questions related to task implementation. After completing the task, the subjects

Table 6: Scorecard given to the subject.

Question	Description	Scores
Q1	I understand the capabilities of this programming language	10
Q2	This language is easy to understand and learn	25
Q3	This language is easy to use and does not require users to have complex programming skills	25
Q4	This programming environment is friendly	15
Q5	This solution provides a quick way to develop I^2B applications	25
Total score		100

Table 7: Design of the experiment.

Stage	Group 1	Group 2	Group 3
Preparation	Introducing three application development methods and training all subjects		
1st phase	C language and Visual Studio	INR and its supporting tool	Touch and its supporting tool
2nd phase	Scoring development methods based on experience		

submit their applications using the teaching platform, which records the development time and number of lines of code for each application. In the second phase, the subjects are asked to score the programming language and programming environment based on their experience and related indicators. Table 8 lists the data generated at each phase of the experiment.

(2) Data Verification. Here, we perform a Shapiro–Wilk W test [53] on each dependent variable (i.e., development time, number of lines of code, and evaluation scores) to check whether the data conformed to a normal distribution. We test with a significance level of $\alpha = 0.05$. The results show that the dependent variables followed a normal distribution. We then use box plots to examine the outliers of the dependent variables, as shown in Figure 15. Since all data samples are located between the lower and upper ends of the box plot, our dataset does not contain outliers.

8.2.3. Experiment Results and Discussion

(1) Data Analysis. Based on the scoring and measurement in Section 8.2.2, the comparison results of Touch with C language and INR are generated, as shown in Figures 16 and 17. Figure 16 shows the average development time, average lines of code, and the average score for developing applications using different methods. The results indicate that the average time to develop applications using C language is the longest, reaching 239.40 minutes. The main reason is that the low level of language abstraction causes the subjects to implement complex low-level details. Although INR introduces some high-level abstractions in the I^2B field to allow developers to complete tasks in an average time of 152.80, Touch performs better. The subjects using Touch complete application development in just 64.00 minutes. In terms of lines of code, the number of lines of code for applications corresponding to the C language reaches 370.80 lines. Compared to this, the number of lines of code corresponding to INR is 133.00 and the number of lines of code corresponding to Touch is only 52.60. When the subjects use C language to program, they need to design a specific algorithm for the problem, resulting in massive amounts of code. INR encapsulates some complex operations into language elements such as operators to reduce the code size to a certain extent. Touch greatly simplifies communication between nodes and further improves the level of abstraction, making the effect more significant. In terms of average scores, C language scored 64.00 and INR scored 72.60, while Touch scored 87.40, which shows that it is easier to develop I^2B APPs with Touch.

Combining the evaluation indicators listed in Table 6, the detailed comparison results of the three development methods are obtained, as shown in Figure 17. The value of Q1 reflects that the subjects understand all three programming languages. In terms of understandability (Q2), Touch gets 21.00 points higher than C language and INR, indicating that Touch is easier to understand and learn. This is mainly because Touch has fewer keywords and terminators and is simpler. Especially in the term of Q3 (ease of use), the C language scores 15.20 and INR scores 18.60. In contrast, Touch got a more satisfactory score of 22.20, which is because Touch adds many domain-specific language elements such as *operators*, *time offsets*, and so on, making the semantic expression stronger. In terms of the friendliness of the programming environment (Q4), the Touch support tool also achieves high scores. The support tool of Touch has a list recommendation function that can automatically display the parameters in SIM for users to choose, which reduces the rate of grammatical errors and the workload of defining variables. For the overall evaluation of programming efficiency (Q5), the advantage of Touch is even more significant, with the highest score of 22.40, while C language only scores 13.8 and INR scores 17.40. In a word, compared with C language and INR, Touch reduces the difficulty of developing I^2B APP and improves development efficiency.

(2) Experimental Results and Discussion. In this section, we discussed the efficiency of Touch. Efficiency depends on the development time and the number of lines of code when using different development methods. When using Touch language to develop applications, compared to C language and INR, the development time is shortened by 73.23% and 58.12%, and the number of lines of code of applications is

Table 8: Experiment data.

	Group 1 (C language)					Group 2 (INR language)					Group 3 (Touch language)				
	S1	S2	S3	S4	S5	S1	S2	S3	S4	S5	S1	S2	S3	S4	S5
Q1	8	10	9	8	9	8	9	9	9	8	9	9	8	9	10
Q2	18	16	18	20	17	19	18	20	16	18	20	23	19	21	22
Q3	14	15	14	16	17	17	19	19	20	18	24	21	23	20	23
Q4	7	9	10	7	9	9	8	10	11	11	12	12	13	13	14
Q5	14	13	15	16	11	17	19	15	20	16	21	23	24	22	22
Total score	61	63	66	67	63	70	73	73	76	71	86	88	87	85	91
Time (min)	255	223	276	232	211	162	148	124	177	153	82	62	70	51	55
Lines of code	374	363	343	383	391	115	152	124	131	143	47	53	59	65	39

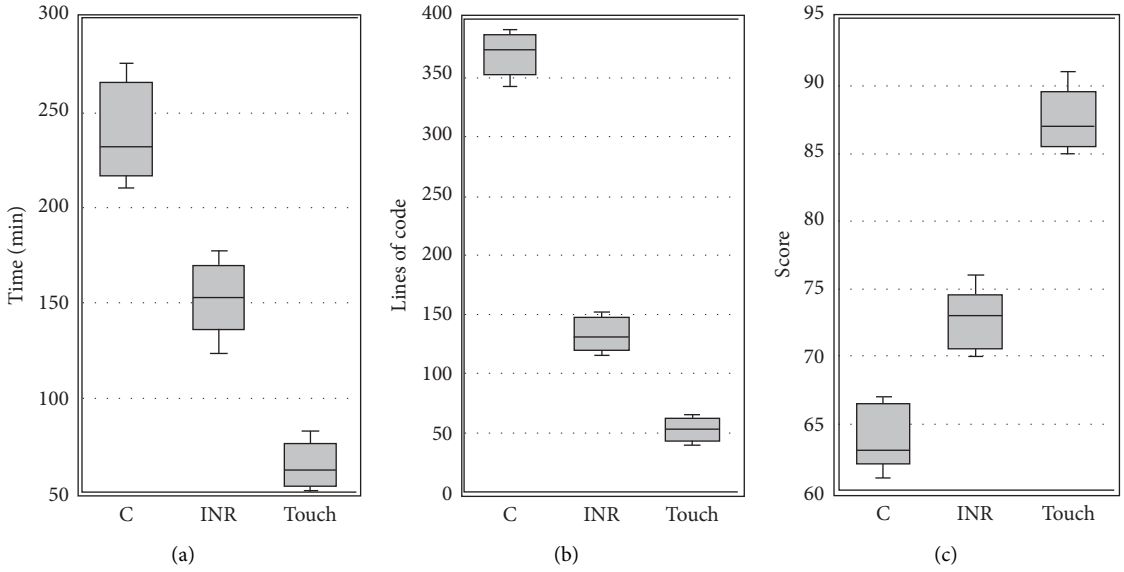


Figure 15: Box plot for outlier analysis. (a) Development time. (b) Lines of code. (c) Total score.

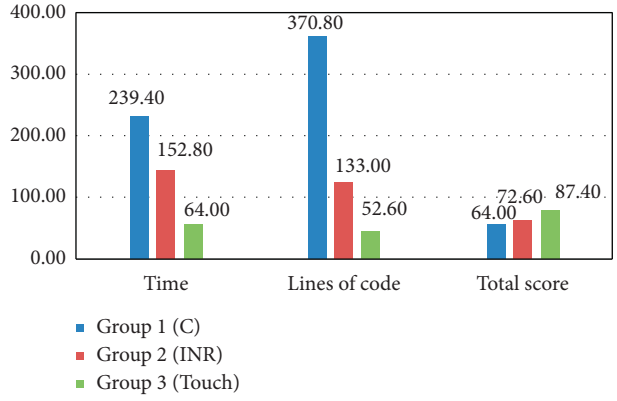


Figure 16: Comparison of development time, lines of code, and total score.

reduced by 85.8% and 60.45% compared to C and INR, respectively. At the same time, the scores of the subjects on different development methods reflect that Touch and its supporting tool are easy to understand and easy to use. It can be seen that Touch has obvious advantages in improving the efficiency of I^2B APP development.

Touch greatly reduces the complexity of grasping the concept of I^2B and effectively promotes the development process of I^2B APP. Since Touch incorporates domain concepts into programming language elements, it provides different levels of abstraction (i.e., *basic units*, *neighborhoods*, and *domains*) and better expressions of complex tasks (e.g., communication between nodes and traversing neighborhood nodes). The Touch support tool provides a friendly development interface, which effectively shortens the development time. Moreover, Touch's syntax is simple and

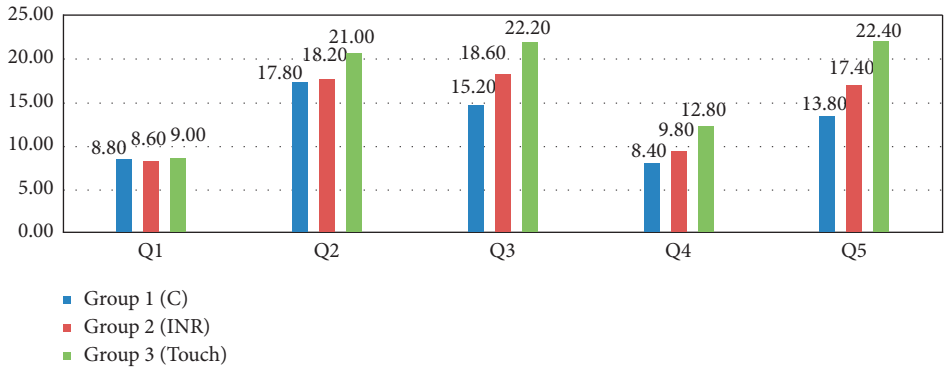


Figure 17: Comparison of the average score to each indicator.

easy to learn and use. Therefore, the Touch language greatly improves development efficiency and reduces the user's development burden.

The issue of DSL evaluation has received increasing attention [54]. We compare the controlled experiment of the Touch language with the controlled experiment of DSL applied to different domains [55–60]. Most of these DSL controlled experiments adopt the method of "implementing the same problem in a DSL and a GPL and comparing the efficiency and accuracy of each approach" [55]. And the design and analysis of our experiments are similar to these DSL studies. On the one hand, the experimental results of these studies show that compared with using GPL, when subjects use DSL, the time to complete the task is significantly reduced [9, 55, 57], the number of lines of code of the program is less [18, 59], and the correctness of the task is higher [55, 57]; at the same time, the subjects' evaluation of the DSL (e.g., ease of use [56, 58], comprehensibility [9, 58], and maintainability of the solution [56, 59]) is also higher. This is similar to the experimental results in this paper: compared with the C language, using Touch language can shorten the development time of I^{2}B APP, reduce the number of lines of code, and improve the satisfaction of the subjects. However, this article does not use the correctness indicator [55] to compare the effectiveness of Touch and C language. In the next step, the correctness will be supplemented as an evaluation indicator in the experimental evaluation stage to enhance scientificity and persuasiveness. Furthermore, the Touch language is not only compared with GPL in this paper but also compared with INR [29] (a DSL) used to I^{2}B APP development in terms of efficiency [60]. The experimental results show that the efficiency of the Touch language is better than INR.

On the other hand, these experiments concluded that using DSL is better than GPL with respect to effectiveness and efficiency [55–59]. This is also similar to the conclusion of this paper: compared with C language and INR, Touch language can effectively develop I^{2}B APP and improve efficiency. However, this paper proves the effectiveness of the Touch language by verifying that the Touch program can be successfully converted to the target code and executed normally on the CPN. Therefore, this paper does not compare the effectiveness of Touch with C language, which is different from some DSL studies [55, 57] that use correctness indicators to evaluate the effectiveness of DSL.

(3) Threats to Validity. In this paragraph, the threat to the validity of this experiment is discussed.

Construct validity: the measures of development time and the number of lines of code in this experiment are standard measures. The concept of development efficiency used in this article is broad, and it is easy to produce construct validity. To reduce this threat, we carefully designed a scorecard on development efficiency, including language understandability, ease of use, and friendliness of development tools. We refined the user experience in the development process into specific projects to reflect the development efficiency of different methods.

Internal validity: in the experiment, we divided the subjects into three groups according to the principle of randomness and balance, and the ability of the subjects in each group was similar. Subjects have never used these methods before. We gave them enough time to question this task to ensure that they understood it. We carefully monitored and avoided plagiarism. No subjects gave up this task.

External validity: external validity represents the universality of the conclusions of this paper. We took the personnel evacuation case as an experimental object, and this task is a typical scenario of I^{2}B applications. The experimental results have universal significance for this type of application. However, the background of the subjects in this paper is limited. In the experiment, we selected a group of experienced graduate students as the experimental object, but Touch is not only designed for students but also designed for I^{2}B engineers. In the future, we plan to conduct more experiments to study the effectiveness and efficiency of Touch and let more I^{2}B engineers participate.

9. Conclusion

I^{2}B is a novel intelligent building platform with excellent flexibility and scalability. To facilitate the development of I^{2}B APPs and reduce the programming difficulty, the textual programming language Touch is proposed. We establish a conceptual programming architecture for I^{2}B APPs containing three key programming concepts: *basic units*, *neighborhoods*, and *domains*. These improve the

intuitiveness of application development. Then, the special language elements are created, including special data types, domain-specific variables, time offsets, etc., to effectively support the parallel computing model and portability aspects of the I^2B platform. The formal definitions for the concrete syntax of Touch are also provided. Moreover, we implement a support tool for I^2B APP development that provides user-friendly GUI support for editing, storing, and managing Touch programs.

In addition, the effectiveness and efficiency of Touch are evaluated using a personnel evacuation APP example. The results show that Touch can effectively support I^2B APP development. Furthermore, compared with the traditional C language and INR programming model, Touch can significantly reduce development time and the number of code lines and has better understandability and ease of use. This advantage occurs because Touch incorporates I^2B domain-specific concepts into Touch language elements and provides direct support for I^2B APP development. Therefore, Touch greatly simplifies the I^2B APP development process and improves user accessibility to applications. Overall, this paper is also a good example that can provide a reference for other research into domain-specific application development.

Our planned future work is twofold. First, we plan to combine the Touch approach with formal methods by analyzing the domain-specific model using a formal method and transforming the artifacts into a formal system. This formal system can then be used to verify practical applications and improve reliability. Second, we plan to investigate the applicability of I^2B application development methods in cyber-physical systems.

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

Special thanks goes to Academician Jiang Yi of Tsinghua University for his guidance and discussion. This research was funded by the National Key Research and Development Project of China (New Generation Intelligent Building Platform Techniques) (2017YFC0704100).

References

[1] S. Wang, *Intelligent Buildings and Building Automation*, Routledge, London, UK, 2009.

[2] P. Domingues, P. Carreira, R. Vieira, and W. Kastner, "Building automation systems: concepts and technology review," *Computer Standards & Interfaces*, vol. 45, pp. 1–12, 2016.

[3] Z. Y. Jiang, "Research on the information flow of building automation system," Ph.D. Dissertation, Tsinghua University, Beijing, China, 2008.

[4] Q. C. Zhao, L. Xia, and Z. Y. Jiang, "Project report: new generation intelligent building platform techniques," *Energy Informatics*, vol. 1, no. 1, pp. 12–16, 2018.

[5] Q. C. Zhao and Z. Y. Jiang, "Insect intelligent building (I^2B): a new architecture of building control systems based on Internet of Things (IoT)," in *Proceedings of the International Conference on Smart City and Intelligent Building*, pp. 457–466, Springer, Hefei, China, September 2018.

[6] Q. Shen, *Studies on architecture of decentralized system in intelligent building*, Ph.D. Dissertation, Tsinghua University, Beijing, China, 2015.

[7] 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.

[8] 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.

[9] T. Kosar, N. Oliveira, M. Mernik et al., "Comparing general-purpose and domain-specific languages: an empirical study," *Computer Science and Information Systems*, vol. 7, no. 2, pp. 247–264, 2010.

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

[11] A. Barišić, V. Amaral, and M. Goulão, "Usability driven DSL development with USE-ME," *Computer Languages, Systems & Structures*, vol. 51, pp. 118–157, 2018.

[12] M. Voelter, S. Benz, C. Dietrich et al., *DSL Engineering: Designing, Implementing and Using Domain-specific Languages*, CreateSpace Independent Publishing Platform, Scotts Valley, CA, USA, 2013.

[13] D. Kamma and K. G. Sasi, "Effect of model based software development on productivity of enhancement tasks—an industrial study," in *Proceedings of the 21st Asia-Pacific Software Engineering Conference*, pp. 71–77, Jeju, South Korea, December 2014.

[14] M. Sulír, M. Bačíková, S. Chodarev, and J. Porubän, "Visual augmentation of source code editors: a systematic mapping study," *Journal of Visual Languages & Computing*, vol. 49, pp. 46–59, 2018.

[15] T. Ugawa, H. Iwasaki, and T. Kataoka, "eJSTK: building JavaScript virtual machines with customized datatypes for embedded systems," *Journal of Computer Languages*, vol. 51, pp. 261–279, 2019.

[16] J. C. Dageförde and H. Kuchen, "A compiler and virtual machine for constraint-logic object-oriented programming with Muli," *Journal of Computer Languages*, vol. 53, pp. 63–78, 2019.

[17] T. Nakamaru, K. Ichikawa, T. Yamazaki, and S. Chiba, "Generating fluent embedded domain-specific languages with subchaining," *Journal of Computer Languages*, vol. 50, pp. 70–83, 2019.

[18] J. Smits, G. Wachsmuth, and E. Visser, "FlowSpec: a declarative specification language for intra-procedural flow-Sensitive data-flow analysis," *Journal of Computer Languages*, vol. 57, Article ID 100924, 2020.

[19] A. A. Bock, T. Bøgholm, P. Sestoft, B. Thomsen, and L. L. Thomsen, "On the semantics for spreadsheets with sheet-defined functions," *Journal of Computer Languages*, vol. 57, Article ID 100960, 2020.

[20] A. Perisic, M. Lazic, and B. Perisic, "The Extensible Orchestration Framework approach to collaborative design in architectural, urban and construction engineering," *Automation in Construction*, vol. 71, pp. 210–225, 2016.

[21] D. Griebler, M. Danelutto, M. Torquati, and L. G. Fernandes, "SPar: a DSL for high-level and productive stream parallelism," *Parallel Processing Letters*, vol. 27, no. 1, Article ID 1740005, 2017.

[22] V. Debusschere, C. Bobineau, Q. Giap et al., "Smartlog—a declarative language for distributed programming in smart grids," *Computers & Electrical Engineering*, vol. 80, Article ID 106499, 2019.

[23] B. Tekinerdogan and E. Arkin, "ParDSL: a domain-specific language framework for supporting deployment of parallel algorithms," *Software & Systems Modeling*, vol. 18, no. 5, pp. 2907–2935, 2019.

[24] S. Nastic, S. Sehic, M. Vögler et al., "PatRICIA—a novel programming model for IoT applications on cloud platforms," in *Proceedings of the International Conference on Service-oriented Computing & Applications*, pp. 53–60, IEEE, Koloa, HI, USA, December 2013.

[25] M. K. Lhaksmana, M. Yohei, and I. T. Ishid, "Role-based programming for implementing adaptive IoT applications," in *Proceedings of the 2016 International Conference on Computer, Control, Informatics and its Applications (IC3INA)*, pp. 179–184, IEEE, Tangerang, Indonesia, October 2016.

[26] M. Zimmermann, B. Uwe, and L. Frank, "A TOSCA-based programming model for interacting components of automatically deployed cloud and IoT applications," in *Proceedings of 19th International Conference on Enterprise Information Systems*, pp. 121–131, Porto, Portugal, April 2017.

[27] B. Cosenza, N. Biagio, B. Juurlink et al., "OpenABL: a domain-specific language for parallel and distributed agent-based simulations," in *Proceedings of European Conference on Parallel Processing*, pp. 505–518, Springer, Turin, Italy, August 2018.

[28] D. Sredojević, M. Vidaković, and M. Ivanović, "ALAS: agent-oriented domain-specific language for the development of intelligent distributed non-axiomatic reasoning agents," *Enterprise Information Systems*, vol. 12, no. 8-9, pp. 1058–1082, 2018.

[29] S. Zhao, Q. L. Yang, J. C. Xing et al., "INR: a programming model for developing APPs of insect intelligent building," *Scientific Programming*, vol. 2020, Article ID 3659849, 2020.

[30] 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.

[31] H. Yu, T. Y. Zhao, and J. L. Zhang, "Development of a distributed artificial fish swarm algorithm to optimize pumps working in parallel mode," *Science and Technology for the Built Environment*, vol. 24, no. 3, pp. 248–258, 2018.

[32] Q. C. Zhao, X. Wang, Y. F. Wang et al., "A P2P algorithm for energy saving of a parallel-connected pumps system," in *Proceedings of the International Conference on Smart City and Intelligent Building*, pp. 385–395, Springer, Hefei, China, September 2018.

[33] S. Wang, J. Xing, Z. Jiang, and J. Li, "Decentralized economic dispatch of an isolated distributed generator network," *International Journal of Electrical Power & Energy Systems*, vol. 105, pp. 297–304, 2019.

[34] R. Membarth, O. Reiche, F. Hannig et al., "Hipacc: a domain-specific language and compiler for image processing," *IEEE Transactions on Parallel and Distributed Systems*, vol. 27, no. 1, pp. 210–224, 2015.

[35] V. Janjic, C. Brown, K. Mackenzie et al., "RPL: a domain-specific language for designing and implementing parallel C++ applications," in *Proceedings of the 2016 24th Euromicro International Conference on Parallel, Distributed, and Network-Based Processing (PDP)*, pp. 288–295, IEEE, Heraklion, Greece, February 2016.

[36] G. Kindlmann, C. Chiw, N. Seltzer et al., "Diderot: a domain-specific language for portable parallel scientific visualization and image analysis," *IEEE Transactions on Visualization and Computer Graphics*, vol. 22, no. 1, pp. 867–876, 2015.

[37] S. Park, Y. C. Lee, and J. K. Lee, "Definition of a domain-specific language for Korean building act sentences as an explicit computable form," *Electronic Journal of Information Technology in Construction*, vol. 21, p. 2016, 2016.

[38] H. Lee, J.-K. Lee, S. Park, and I. Kim, "Translating building legislation into a computer-executable format for evaluating building permit requirements," *Automation in Construction*, vol. 71, pp. 49–61, 2016.

[39] M. Alves, P. Carreira, and A. A. Costa, "BIMSL: a generic approach to the integration of building information models with real-time sensor data," *Automation in Construction*, vol. 84, pp. 304–314, 2017.

[40] J.-K. Lee, C. M. Eastman, and Y. C. Lee, "Implementation of a BIM domain-specific language for the building environment rule and analysis," *Journal of Intelligent & Robotic Systems*, vol. 79, no. 3-4, pp. 507–522, 2015.

[41] E. Grabska, A. Łachwa, and G. Ślusarczyk, "New visual languages supporting design of multi-storey buildings," *Advanced Engineering Informatics*, vol. 26, no. 4, pp. 681–690, 2012.

[42] X. Nguyen, H. T. Tran, H. Baraki et al., "FRASAD: a framework for model-driven IoT application development," in *Proceedings of the 2nd World Forum on Internet of Things (WF-IoT)*, pp. 387–392, IEEE, Milan, Italy, December 2015.

[43] L. Riliskis, J. Hong, and P. Levis, "Ravel: Programming IoT applications as distributed models, views, and controllers," in *Proceedings of the 2015 International Workshop on Internet of Things towards Applications*, pp. 1–6, Seoul, Republic of Korea, November 2015.

[44] J. Verriet, L. Buit, R. Doornbos et al., "Virtual prototyping of large-scale IoT control systems using domain-specific languages," in *Proceedings of the 7th International Conference on Model-Driven Engineering and Software Development*, pp. 229–239, SCITEPRESS-Science and Technology Publications, Prague, Czech Republic, February 2019.

[45] A. J. Salman and A. Al-Yasiri, "Developing domain-specific language for wireless sensor network application development," in *Proceedings of the 11th International Conference for Internet Technology and Secured Transactions (ICITST)*, pp. 301–308, IEEE, Barcelona, Spain, December 2016.

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

[47] E. Visser, R. Lämmel, J. Visser, and J. Saraiva, "WebDSL: a case study in domain-specific language engineering," *Lecture Notes in Computer Science*, vol. 5235, pp. 291–373, 2008.

[48] W. Frakes, R. Prieto-Diaz, and C. Fox, "DARE: Domain analysis and reuse environment," *Annals of Software Engineering*, vol. 5, pp. 125–141, 1998.

[49] D. Weiss and C. T. R. Lay, *Software Product Line Engineering*, Addison-Wesley, Boston, MA, USA, 1999.

[50] M. Eysholdt and H. Behrens, "Xtext: implement your language faster than the quick and dirty way," in *Proceedings of the 25th ACM International Conference Companion on*

Object-Oriented Programming, pp. 307–309, Systems, Languages, and Applications Companion, Reno/Tahoe, Nevada, October 2010.

[51] L. Bettini, *Implementing Domain-specific Languages with Xtext and Xtend*, Packt Publishing Ltd., Birmingham, UK, 2016.

[52] Q. Zhou, J. Xing, and Q. Yang, "Device-free occupant activity recognition in smart offices using intrinsic Wi-Fi components," *Building and Environment*, vol. 172, Article ID 106737, 2020.

[53] P. Royston, "Approximating the shapiro-wilk W-test for non-normality," *Statistics and Computing*, vol. 2, no. 3, pp. 117–119, 1992.

[54] A. Barišić, V. Amaral, and M. Goulão, "Usability evaluation of domain-specific languages," in *Proceedings of the 2012 Eighth International Conference on the Quality of Information and Communications Technology*, pp. 342–347, IEEE, Lisbon, Portugal, September 2012.

[55] T. Kosar, M. Mernik, and J. C. Carver, "Program comprehension of domain-specific and general-purpose languages: comparison using a family of experiments," *Empirical Software Engineering*, vol. 17, no. 3, pp. 276–304, 2012.

[56] A. Johanson and W. Hasselbring, "Effectiveness and efficiency of a domain-specific language for high-performance marine ecosystem simulation: a controlled experiment," *Empirical Software Engineering*, vol. 22, no. 4, pp. 1–31, 2017.

[57] A. Barišić, V. Amaral, M. Goulão et al., "Quality in use of domain-specific languages: a case study," in *Proceedings of the 3rd ACM SIGPLAN Workshop on Evaluation and Usability of Programming Languages and Tools*, pp. 65–72, Portland, Oregon, October 2011.

[58] M. Jiménez, F. Rosique, P. Álvarez, and A. Iborra, "Habitation: a domain-specific language for home automation," *IEEE Software*, vol. 26, no. 4, pp. 30–38, 2009.

[59] J. R. Hoyos, J. García-Molina, and J. A. Botía, "A domain-specific language for context modeling in context-aware systems," *Journal of Systems and Software*, vol. 86, no. 11, pp. 2890–2905, 2013.

[60] F. Haeser, M. Felderer, and R. Breu, "Is business domain language support beneficial for creating test case specifications: a controlled experiment," *Information & Software Technology*, vol. 79, pp. 52–62, 2016.