FLEXIBO

LANGUAGE AND ITS APPLICATION TO STATIC ANALYSIS

Thesis submitted for the degree of
Doctor of Philosophy
at the University of Leicester

by

Jianguo Zhou BSc
Department of Computer Science
University of Leicester

December 2006
To my parents.
# Table of Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Table of Contents</td>
<td>iv</td>
</tr>
<tr>
<td>Abstract</td>
<td>vii</td>
</tr>
<tr>
<td>Acknowledgements</td>
<td>viii</td>
</tr>
<tr>
<td>Introduction</td>
<td>1</td>
</tr>
<tr>
<td>0.1 The Language-FLEXIBO</td>
<td>1</td>
</tr>
<tr>
<td>0.1.1 Why is it &quot;interpreted&quot;?</td>
<td>2</td>
</tr>
<tr>
<td>0.1.2 Why do we need this interpreted language?</td>
<td>3</td>
</tr>
<tr>
<td>0.1.3 What identifies FLEXIBO from other flexible languages?</td>
<td>3</td>
</tr>
<tr>
<td>0.2 FLEXIBO and Rapid Prototype Development</td>
<td>3</td>
</tr>
<tr>
<td>0.2.1 Language Suitability for RAD</td>
<td>4</td>
</tr>
<tr>
<td>0.2.2 Why is FLEXIBO suitable for RAD?</td>
<td>7</td>
</tr>
<tr>
<td>0.3 FLEXIBO and Abstract Interpretation</td>
<td>8</td>
</tr>
<tr>
<td>0.3.1 FLEXIBO and BSP program translation</td>
<td>9</td>
</tr>
<tr>
<td>0.3.2 What is the use of static analysis in prototyping?</td>
<td>10</td>
</tr>
<tr>
<td>0.3.3 What make FLEXIBO more suitable for this kind of applications?</td>
<td>11</td>
</tr>
<tr>
<td>0.4 Thesis structure</td>
<td>12</td>
</tr>
<tr>
<td>1 Literature Review</td>
<td>13</td>
</tr>
<tr>
<td>1.1 Introduction</td>
<td>13</td>
</tr>
<tr>
<td>1.2 Class-based languages</td>
<td>14</td>
</tr>
<tr>
<td>1.2.1 Smalltalk</td>
<td>14</td>
</tr>
<tr>
<td>1.2.2 C++</td>
<td>16</td>
</tr>
<tr>
<td>1.2.3 Eiffel</td>
<td>17</td>
</tr>
<tr>
<td>1.2.4 Java</td>
<td>19</td>
</tr>
<tr>
<td>1.3 Object-based languages</td>
<td>20</td>
</tr>
<tr>
<td>1.3.1 Self</td>
<td>21</td>
</tr>
<tr>
<td>1.3.2 Cecil</td>
<td>22</td>
</tr>
<tr>
<td>1.4 Summary</td>
<td>24</td>
</tr>
</tbody>
</table>
2 FLEXIBO Language Overview

2.1 FLEXIBO’s instructions ................................................................. 26
  2.1.1 Primitive expressions ............................................................... 26
  2.1.2 Composite Expressions ............................................................ 26

2.2 Class and Method
  2.2.1 Class ......................................................................................... 29
  2.2.2 Method .................................................................................... 33

2.3 FLEXIBO’s features ........................................................................ 35
  2.3.1 Operator overloading .................................................................. 35
  2.3.2 Exception .................................................................................. 38
  2.3.3 Ownership and control ............................................................... 40
  2.3.4 Language interaction ................................................................. 43
  2.3.5 Colorful Variable ...................................................................... 44
  2.3.6 Flexible Attribute Binding ......................................................... 46
  2.3.7 Type Constraint ........................................................................ 51

2.4 Summary .......................................................................................... 52

3 Object-oriented Language Interaction .................................................. 54

3.1 Introduction ...................................................................................... 54

3.2 Class, Object and Method
  3.2.1 Interaction Strategy .................................................................... 57
  3.2.2 Interaction between FLEXIBO and Java ....................................... 60

3.3 Cross-language Inheritance
  3.3.1 Cross-language Inheritance from a class ...................................... 65
  3.3.2 Cross-language inheritance from an interface ......................... 67
  3.3.3 Interaction between FLEXIBO and Java ....................................... 69

3.4 Related work .................................................................................... 75

3.5 Summary .......................................................................................... 76

4 Generating BSPlib-C code from LOGS Specifications(I) ......................... 78

4.1 Introduction ...................................................................................... 78

4.2 Syntactical checking ......................................................................... 83

4.3 Basic assumptions, normal form and completeness ............................ 88

4.4 Detecting communication interference .............................................. 92

4.5 Type checking ................................................................................. 98

4.6 Variable registration for communication ........................................... 102

4.7 Analysis of communication dependencies ....................................... 103

4.8 Generating code for expressions ...................................................... 107

4.9 Generating code for communications .............................................. 108

4.10 Generating code for processes ....................................................... 109

4.11 Summary ....................................................................................... 109
Abstract

This thesis introduces a new object-based language FLEXIBO to support prototype development paradigm and more importantly, program static analysis. FLEXIBO offers extreme flexibility and hence enables developers to write programs that contain rich information for further analysis and optimization. FLEXIBO interpreter’s seamless integration with Java (including direct access to Java classes and methods and direct inheritance of Java classes) makes it a suitable tool for fast prototype software development. FLEXIBO’s extreme flexibility allows developers to redefine the behavior of program evaluation by overriding its default evaluation method. This mechanism can be used to translate FLEXIBO to other efficient languages. In this thesis we design a translator in FLEXIBO to translate Bulk-Synchronous Parallel specifications (expressed in FLEXIBO) to executable C programs linked with BSPLib. Before translation, the tool first checks syntax and type, then statically analyzes potential communication conflicts, and finally generates C code. The translation process can accurately analyze primitive commands but require approximation (using abstract interpretation) for more advanced commands such as loops. The appropriateness of the translator and the associated static analysis can be formally analyzed using the technique of normal form.
I would like to express my deep gratitude to Dr. Yifeng Chen, my supervisor, for constant help, guidance and encouragement through the past years. His discussions and suggestions are invaluable. Without his support, the thesis would not exist.

I am also grateful to all the members in the department of computer science at the university of Leicester. Without their help and support, my study and life would be much harder in the foreign country.

I would like to thank my parents and Wenting for their patience and love. Without them, this work would never have come into existence.

Finally, I feel grateful to the following people: Zhiming Liu, Dekun Yang, Shengxiang Yang, Rui Cai and Zhidong Jia. Their constant encouragement helps me over the last few years.

Jianguo Zhou
Introduction

Object-oriented programming is the most influential development paradigm in the last decades. Unlike the traditional ones[29, 51, 95], it defines data structure to incorporate both data type and applicative functions. The most important benefit comes from its reusability and extendibility [81]. Object-oriented programming languages have been successfully applied in many areas such as industry [63], education [49], etc. However, as far as rapid prototype development(RAD) [83] is concerned, most of them are not satisfactory. Compilation-based languages have too many restrictions to affect rapid development and testing. In contrast, although interpreted languages have powerful expressibility and flexibility, their performance is normally far from satisfactory.

The design objective of FLEXIBO is to produce such a new object-oriented language that does not only provide extreme flexibility for rapid development, but also have a reasonable approach to improve programming efficiency. FLEXIBO allows developers to present abstract and flexible programs, and translate its programs to efficient languages(eg. C++). Although FLEXIBO does not have high efficiency, its corresponding programs can efficiently run in target languages. From this perspective, the translation can be considered as a way to “increase” programming performance.

0.1 The Language-FLEXIBO

FLEXIBO is a flexible interpreted object-oriented language that provides a decentralized development environment for developers with different levels of trust. This language pursues extreme flexibility for programming development. It regards primitive, objects, classes and methods as first-class values, and allows developers to redefine many language features such as classes, methods and even
method execution through inheritance. It is also able to check accessibility, dynamic constraints at run-time.

**FLEXIBO** is designed to be a browser-based language with the support of Java virtual machine [59, 64]. Developers do not need to install its interpreter on client computer. Since Java applet can safely run on client side, the online cooperation is secure. Server computer can effectively defense various “undesired” behaviors such as resource(eg. memory) over-consumption and non-terminated execution.

This language can easily use Java’s properties. Developers are allowed to declare Java classes, instantiate Java objects and invoke Java methods, even inherit classes from Java. All the operations naturally incorporate with **FLEXIBO**. Developers do not need to know the difference between these languages.

### 0.1.1 Why is it “interpreted”?

Why do we need an interpreted language? To maximize performance, compilation-based languages decide most features statically, and normally lack flexibility. For instance, C++ [28] binds its attributes\(^1\) at compilation, does not naturally support dynamic dispatching [32, 33]. By comparison, interpreted languages provide more flexibility. They delay most compilation work to program execution, and determine those language features at runtime. It is much easier for these languages (eg. Ruby [88]) to present dynamic attribute binding.

Moreover, interpreted languages allow rapid development and testing at the same time. They do not particularly separate compilation from programming execution. Programs can be tested as they are being developed. Therefore, some program errors can be removed as they are coded. On the contrary, for compilation languages, testing normally happens after program development.

\(^1\) An attribute can be either a field or a method.
0.1.2 Why do we need this interpreted language?

After all, there exist many object-oriented programming languages. Firstly, FLEXIBO can be used as an usual language. From this perspective, it is an interpreted programming language with a few novel mechanisms, which make this language more easy-to-use. Secondly, FLEXIBO is more suitable for rapid prototype development than most conventional languages because of its extreme flexibility and pure object-orientation. Finally, this language allows developers to redefine the behaviors of language instructions. A common program may produce different results under various redefinitions. For instance, in order to improve performance, a redefinition may translate FLEXIBO's programs to efficient languages. From FLEXIBO's perspective, the process occurs at run-time. But it happens before its execution in target languages.

0.1.3 What identifies FLEXIBO from other flexible languages?

Real flexible languages do not only provide freedom to developers, but also allow them to define desired constraints. FLEXIBO's design philosophy is to provide an extremely flexible meta-language, and allows developers to add individual constraints, therefore produces a set of sub-languages with different constraints. For instance, this dynamically typed language does not reserve many type restrictions. But developers can strengthen more type constraints through redefining the execution for a predefined method.

0.2 FLEXIBO and Rapid Prototype Development

Rapid prototype development (RAD) is a development paradigm (see Figure 1) used to help customers and developers accurately understand system requirements [83]. Involving prototype development enables customers to clearly see how the developing system supports their work. RAD produces overall impression of software product, and makes easier to reveal errors and omissions in system requirements.
In particular, RAD provides the following benefits. Firstly, it presents early requirement definition, refinement and validation. It allows customers to remove undetected and confusing services, therefore reduces project requirement risks. Secondly, RAD encourages customers to involve project development. Rather than complicated development details, prototype product normally provide more straightforward interaction for customers. Thirdly, available prototypes support user training and system testing. Users do not need to take these procedures until systems are fully completed. Finally, RAD normally provides a general framework for later system development [83].

0.2.1 Language Suitability for RAD

How do we evaluate language suitability for RAD? Generally, they need to provide great flexibility, clean and simple syntax, powerful expressivity. In detail, the following principles are proposed to examine suitability for RAD. These requirements may be needed by other kinds of applications, but are highly important for this paradigm.

Clean concept and no ambiguity To quickly produce feedback, prototype programs are normally
abstract and expressive. To guarantee program readability, RAD languages need to provide clean concepts, and minimize program ambiguity. For instance, dynamic dispatching effectively reduces code redundancy, but many languages (e.g., Java) do not clearly show developers how to apply this mechanism on different attributes [15, 48].

**Pure object-orientation** Object-oriented programs offer great reusability and extendibility [11]. Many languages claim to support pure object-orientation. However, the justification highly relies on the standards. The following rules are used to evaluate this feature.

- **Information hiding.** A pure object-oriented language is able to encapsulate states and provide multiple-level access.
- **Polymorphism.** This mechanism encourages developers to write more expressive and flexible programs. It is also helpful to reduce code redundancy.
- **Types as objects.** It requests both user-defined types and predefined types to be regarded as normal objects.
- **Messaging.** A pure object-oriented programming language treats all the operations, even arithmetic through the way of message passing.

**High level programming** Languages need to provide some features to support high-level programming.

- These languages should manage memory allocation and release. Ideally, automatic garbage collectors need to be provided.
- Memory-based pointer operations are forbidden in order to avoid misusing.

**Clean syntax** Language syntax is not only closely related to program understandability, but also software maintaining. Readable syntax helps developers to quickly understand legacy system. The following principles are used to examine language syntax.
• Unified operators. Similar or same operations should use common operators.

• Intuitional syntax. Ideal syntax makes developers clearly understand keywords, labels and operators. For instance, := as assignment is more straightforward than = since the latter may be mistaken as equality.

• Similar to widely used languages. More similarity makes learners quickly master languages.

Reduction of redundant code Code redundancy seriously affects software quality. We try to use language flexibility to minimize redundant code while does not affect program readability. An ideal language is expected to reduce the following issues.

• Duplicated code. A piece of common code repetitively appears in a superclass and its subclasses, and can be simplified through dynamic dispatching.

• Similar code. The minor differences come from the denotations of certain attributes.

Interaction between languages RAD languages pursue great flexibility and expressivity, but do not need to support all language features. If they can naturally interact with other languages, we also regard them to possess the properties of target ones. The following principles are used to check language interaction.

• Natural interaction. Languages manage interaction process, which does not need the involvement of developers.

• No additional instructions. Cross-language instructions should reuse the existing ones in local languages.

Easy-to-learn and easy-to-use To quickly learn and master them, these languages should be easy to learn and easy to use. We present the following rules to check whether languages meet this requirement.
• Simplification and unification. An ideal language unifies similar operations through common approaches. It prevents developers from misusing and misunderstanding similar concepts.

• Similar to other languages. They should use widely used instructions and their semantics. The similarity helps developers to quickly learn and minimize potential ambiguity.

Most influential object-oriented programming languages do not present all the properties above. The pure object-oriented language-Smalltalk [34] has a special syntax, which is sharply different with widely used ones. C++ [9] regards efficiency as the most important goal, and tries to eliminate any overhead that object-oriented features may cause. It is not a pure object-oriented programming language, and forces developers to carefully handle low-level features (eg. memory release). Java has an ad-hoc attribute binding mechanism. It determines fields at compilation, but methods at runtime. The lack of unification may affect program understandability. Other languages including Self [82] do not support RAD easily.

0.2.2 Why is FLEXIBO suitable for RAD?

FLEXIBO is a pure object-oriented language with extreme flexibility to support RAD. To produce flexible prototype programs, this language hides most low-level operations (eg. memory management), and does not support pointer-based calculus. To maximize its flexibility and reduce ambiguity, it uses an unified approach to determine all its attributes. This language regards all the first-class values, even classes and methods as objects. Its unified object model reserves a single-root hierarchy. All the classes are derived from the root Value. This language exhibits clean and simple syntax. It does not only try to be compatible with other languages in syntax, but also removes some confusing instructions and mechanisms in those languages. To reduce code redundancy and enhance expressivity, this language provides a flexible binding approach. It allows developers to indicate explicitly expected attributes and their binding approaches. Furthermore FLEXIBO can interact with Java naturally. With a few extra instructions, this language enables developers to freely import
Java classes, create Java objects, even inherit from Java classes and interfaces. The interaction is transparent to developers, does not need developers' involvement.

### 0.3 FLEXIBO and Abstract Interpretation

Abstract interpretation (AI) [24, 25] is the technique that automatically analyzes program properties without execution. This technique rarely generates accurate values, but approximates their ranges or domains through evaluating programs statically. For instance, we abstractly analyze the values of the multiplication of two integers. An integer can be either odd or even. Assume that o stands for any odd integer, e for any even integer. Only the multiplication of two odd integer produces an odd integer, otherwise an even one. AI-based calculation produces either o or e as output instead of an accurate result.

<table>
<thead>
<tr>
<th></th>
<th>o</th>
<th>e</th>
</tr>
</thead>
<tbody>
<tr>
<td>o</td>
<td>o</td>
<td>e</td>
</tr>
<tr>
<td>e</td>
<td>e</td>
<td>e</td>
</tr>
</tbody>
</table>

In practice, this technique has been applied on many areas. Firstly, this technique is helpful to type inference [22], which relies on types rather than concrete values. Secondly, it can be used to program optimization [46]. Abstract interpretation is also used to discover some unreachable reductant code and check the freedom of run-time errors [52]. Finally, AI benefits program translation [45].

**FLEXIBO** provides an important mechanism named *reflection*, which allows developers to reuse and redefine its properties (e.g., evaluation). A source program may be interpreted differently under different reflection systems. In general, the normal evaluation can be regarded as the default predefined semantics of FLEXIBO programs. This mechanism benefits from easily presenting additional property checking. For instance, developers can design a reflection system to approximately check program termination.
0.3.1 FlexibO and BSP program translation

Bulk-Synchronous Parallelism [91] is a programming paradigm based on variable sharing and global synchronization. In BSP, processes are synchronized at corresponding synchronization commands issued by individual processes. Local computation commands appear between consecutive synchronization. Most communications are delayed till the following synchronization point at which their delivery is guaranteed. Synchronization points partition the execution of any BSP program as a series of supersteps. BSP has a simple model for complexity analysis. However the main challenge still lies in parallel program development [94].

Typically BSP programs are developed in the SIMD style. One program runs multiple processors, each of which may process a segment of the input data. The length and relative position of each segment in a large array must be calculated dynamically by the parallel program according the dynamic process id. Such code can be difficult to write, as the size of the input data may not be exactly aligned to a particular partitioning method.

In fact, BSP programming [40] can be presented as MIMD-style. For example, we can store different C procedures (as function pointers) in an array and use a single program to call one of them according to the dynamic process id number. This is essentially MIMD programming: different processors run different program procedures in execution.

A BSPlib-C program is a normal C program with BSPlib [40] that provides a set of functions. The functions bsp_pushregister and bsp_popregister register and release a piece of shared memory for communications respectively. The function bsp_sync synchronizes with other processes' bsp_sync commands. The function bsp_put are used to send some data from a local address to an address on a remote process. bsp_get requests some date from an address on a remote process.

Chen and Sanders [19] design an intermediate specification language LOGS for MIMD program development. It makes explicitly the intermediate global states at synchronization points. A certain
form of specifications can be transformed into program code. Chen and Sanders [18] present refinement rules from LOGS to a simplified BSP language and introduce a pair of commands for variable protection. Communication commands are inserted into specification through the refinement laws, and communication interference can be detected with algebraic laws of variable protection.

We try to apply AI and FLEXIBO's reflection mechanism to design a tool to automatically translate FLEXIBO-based specifications to executable C programs linked with BSPlib. Before generating target code, the translator needs to check the syntax, freedom of communication interference, type consistency and communication dependencies between processes. The approach builds on abstract interpretation [20, 21].

Abstract interpretation is a theoretical foundation of static-analysis methods based on denotational semantics. For example, syntactical checking can be defined as an abstract boolean function on program constructs. Code generation becomes an abstract function that transforms a program into a string in a target language. In general, compilation can be considered as a special translation whose target language is a low-level one.

Each translation corresponds to a reflection system consisting of the classes that override FLEXIBO's own classes of program constructs. User-defined evaluation method replaces the pre-defined method to evaluate the reflected programs. A common FLEXIBO program may be converted into several programs under different reflection systems. In this case, we need to design several systems for syntax checking, type consistency and so on.

0.3.2 What is the use of static analysis in prototyping?

Static analysis corresponds to the formal technique abstract interpretation, which can be used to formally study programming properties. In many cases, we do not expect prototype programs to produce rapid feedback, but also reflect some properties (eg. correctness and termination). The thesis demonstrates how to statically validate whether the access to an array is out of index. Moreover, static analysis can be used to simulate the features that FLEXIBO does not directly provide.
One of the typical examples is the simulation of type checking. FLEXIBO does not support static type checking. Type inconsistency triggers run-time exceptions. With static analysis, this work can be similarly completed before program execution. Code transformation is another typical application of static analysis. Although prototype programs are expressive, they do not possess high efficiency. Code transformation is an effective approach to guarantee both flexibility and preformation. The thesis shows the feasibility to transform prototype programs (i.e. LOGS specifications) to corresponding C code.

0.3.3 What make FLEXIBO more suitable for this kind of applications?

There mainly exist the following factors. Firstly, the unusual flexible language eliminates many syntactical restrictions that commonly exist in many languages, therefore enables developers to present abstract and expressive specifications. By comparison, most widely used languages do not reserve such capability. Secondly, FLEXIBO enables developers to define desirable constraints. Unlike FLEXIBO, LOGS has many restrictions in both syntax and semantics. How to present these desirable constraints is the key requirement to freely write LOGS programs. In this case, we define a small sub-language with more restricted syntax based on FLEXIBO. This redefinition is straightforward and natural, does not force developers to involve in the complicated process for either lexical or syntactical checking. Thirdly, the mechanism reflection is presented through a pure object-oriented style. Object-oriented programming possesses great reusability and extendability. FLEXIBO enables developers to override the behavior of program evaluation with object-oriented manner. It reserves inheritance relation in any reflection system. Therefore, developers only need to redefine the instructions expected to be overridden, others are able to reuse the behavior through inheritance. Finally, FLEXIBO is able to guarantee both flexibility and preformation. LOGS-based programs may be highly abstract to express various parallel programs. They will not be executed in LOGS or FLEXIBO platform. On the contrary, the specifications are transformed to BSPLib-linked C code, then efficiently run in that target environment.
0.4 Thesis structure

Chapter 2 evaluates the suitability of some influential object-oriented programming languages for RAD. The third chapter overviews FLEXIBO and its important features including "colorful" variables, flexible binding approaches and so on. The next chapter proposes a solution for natural interaction between FLEXIBO and other languages (eg. Java). Chapter 5 designs and implements a tool, which translates an abstract specification to its corresponding C++ program. It also formally proves the correctness of this translation process. The sixth chapter extends the work in Chapter 5 through adding more expressive instructions, and formally validates the translation process. The next chapter defines the formal operational semantics, which accurately describes FLEXIBO's execution details. The final chapter is given to make conclusions and discuss the further work.
Chapter 1

Literature Review

1.1 Introduction

The first object-oriented programming language (OOPL) - Simula [55, 73] emerges in the 1960s. It built on Algol-60 [80] supports objects, single inheritance, abstract classes, method overloading and nest structures. It almost presents all the key features that this kind of languages possess, but lacks the support to dynamic dispatching [86]. Smalltalk [53, 54] is the first widely used OOPL. This language firstly introduces the term- *object orientation*. In order to maximize programming expressivity and flexibility, this dynamically typed language allows flexibly determining attributes. C++ [85] and Java [37] are the most widely used languages. The former extends the language C [51] with some object-oriented features. C++ is expected to be totally compatible with C and eliminate the overhead that the extra features may cause. Java benefits programming development from removing a number of easy-to-confuse notions and features, and provide an automatic garbage collector [26] to handle spare objects. Object-based languages such as Self [90] and Cecil [12] have simple structure, and provide great expressivity and flexibility.

In the last decades, OOPLs are widely used in industry [85, 63], education [49], research [16, 30]
and other sectors. This chapter reviews some influential OOPLs and examines their suitability to support RAD [83]. To clearly evaluate these languages, this chapter separately discusses class-based ones and object-based ones [1]. In class-based languages, a class represents an abstract data type, describes the common properties (e.g., behaviors) for its objects. Objects are real entities. In contrast, object-based languages focus on objects themselves. An object can be regarded as a template. To produce a new object is to clone the template.

1.2 Class-based languages

Class-based languages constructs the mainstream of OOPLs. The well-known class-based programming languages include the first OOPL Simula, the first widely used one Smalltalk, the most widely used ones: C++ and Java.

1.2.1 Smalltalk

Smalltalk [34] emerged in the early 1970s. To improve flexibility and expressivity, it was designed as a dynamically typed language. Therefore, developers do not need to specify types when variables are declared. This language describes all the operations through message-passing [86]. Smalltalk supports single inheritance, abstract classes and method overloading. The aim to provide inheritance is not only to reduce redundant code and improve reusability, but also to specialize objects. An object of a subclass can be safely assumed as the object of its superclasses [34].

Is Smalltalk suitable for RAD paradigm? We evaluates its suitability from the following aspects.

Clean concept and no ambiguity Smalltalk clearly presents most features, still remains a few points that may cause ambiguity. For instance, this language does not provide object constructors. To initialize an instance, developers must declare some particular methods. Since these methods are not standard, developers may misuse them.

Pure object-orientation Smalltalk regards object as the most fundamental concept. Any value can
be defined as an object. This language even represents statement execution with message-passing style. Smalltalk encapsulates all its data into either objects or classes and supports multilevel access control. As a dynamically typed language, Smalltalk naturally supports polymorphism. The class `Object` is the root of its whole class hierarchy. All operations including arithmetic ones are based on message passing.

**High level programming** In Smalltalk, developers do not deal with low-level operations. This language provides a built-in garbage collector [34] to automatically collects spare objects and releases the allocated memory. Smalltalk hides pointer-oriented calculations.

**Clear syntax** Smalltalk presents a special syntax, which is sharply different from most OOPL. There are various options [49] about the syntax. We do not think that the syntax is ideal. It speciality hinders developers experienced in other languages to easily understand programs.

**Reduction of redundant code** Smalltalk naturally supports polymorphism. This language applies dynamic dispatching to flexibly determine methods, therefore reduces code redundancy and improves reusability.

**Interaction between languages** Smalltalk can interact with the language C, but need developers' involvement. For instance, developers need to specify how to map argument objects from data types in C to their Smalltalk interpreter [10]. Moreover, Smalltalk defines a set of keywords like `SystemDictionary` to particularly support the interaction.

**Easy-to-learn and easy-to-use** Smalltalk is an easy-to-use language. It provides an unified object model to represent all its values including classes and methods as objects. The property-dynamic dispatching benefits developers from presenting simple and flexible programs. However, due to its particular syntax, Smalltalk are not easy-to-learn, especially to the experienced developers in other languages.
1.2.2 C++

C++, as the extension of C was released in the early 1980s [85]. It keeps compatible with the original language and adds a few object-oriented features. C++ allows developers to control many low-level facilities (e.g., memory). Its additional object-oriented support includes virtual functions, access control to individual attributes, friend classes, nested classes and multiple inheritance. Since this language regards efficiency as its key goal, it does not support dynamic dispatching by default [58]. To dynamically bind a method, developers must use virtual methods. Moreover, this language allows a class to inherit multiple classes and support generic classes through the mechanism-template [8].

We examine its suitability for RAD from the following aspects.

**Clean concept and no ambiguity** C++ does not clearly present some concepts. To guarantee the compatibility with C and eliminate the overhead of object-orientation, it remains a few easy-to-confuse mechanisms. For example, C++ allows assigning a value to a function pointer.

**Pure object-orientation** C++ are not a pure object-oriented programming language. It possesses the features of both procedural languages and object-oriented ones. The hybrid language does not directly support polymorphism, but partially through virtual methods. It does not regard predefined types as objects. For instance, all the primitive types (e.g., integer) are not in the world of objects. Therefore the operations on primitive values are based on message passing.

**High level programming** C++ originates from C and attempts to accept all programs code in C. Therefore it does not hide low-level operations (e.g., pointer arithmetic). To eliminate potential overhead, C++ does not provide an built-in garbage collector. Developers must deal with memory allocation and release manually [50].

**Clear syntax** In general, C++ exhibits a readable syntax. Nevertheless, due to its confusing pointer-based operations, some strange expressions harm readability. The examples includes the assignment to function pointers, pointer type cast, etc.
Reduction of redundant code  C++ tries to maximize efficiency. Reducing code redundancy is not its key objective. In some cases, the improvement in efficiency may sacrifice programming readability.

Interaction between languages  C++ can easily interact with C, and accepts all the programs coded in C. It is also able to embed the code of assembly languages [51]. This mechanism is used to fasten executing pieces of frequently used code. Evaluated with the principles, it does not present a friendly interaction, but affects program readability and understandability.

Easy-to-learn and easy-to-use  C++ is not an easy-to-learn language. To master it needs to known many low-level issues about computer system. For instance, to properly use the mechanism-pointer, they must cleanly understand when and how computers allocate and release memory. There exist two different views as to whether this language is easy-to-use. C++ does not completely support pure object-orientation. Properly using this language is challenging. However, some developers may regard it as easy-to-use because it allows developers to control low-level operations freely. As far as RAD is concerned, we do not regard C++ as an ideal language because of its complication.

1.2.3  Eiffel

Eiffel [60] was released in the 1980s. This language provides a powerful, object-oriented integrated design environment, and supports many aspects of modern software engineering [63]. In general, Eiffel has the following features. Firstly, it considers that “Type” and “Class” are identical. Type equivalence relies on class equivalence. Secondly, Eiffel presents multiple inheritance and a renaming mechanism to handle multiple attributes to share common names. Finally, it provides the built-in support to “Design By Contract” [62], which is the mechanism to effectively debug, test and assume software quality [63].

Clean concept and no ambiguity  Eiffel has two kinds of storages to record objects. The first one
is like the usual approach to represent objects through references. The second one is named expanded [63], which completely stores objects rather than their references. Through this paradigm, Eiffel can regard everything as an object and does not need to separately handle primitive values from objects. To determine attributes, Eiffel uses dynamic binding for both fields and methods.

**Pure object-orientation** Eiffel is a pure OOPL, and easily presents polymorphism. All its operations are based on message passing. It supports multiple inheritance through its novel renaming mechanism [60]. Eiffel allows developers to explicitly identify accessibility for different classes [63].

**High level programming** This language provides a garbage collector to automatically release memory for spare object, and does not support pointer-based computation.

**Clear syntax** Eiffel provides an ideal syntax, which is more readable than most widely used OOPL. This language does not use the symbol-semicolon to separate statements. Each of them stands a line. This style does not appears in other languages.

**Reduction of redundant code** Its flexible mechanism in attribute binding greatly benefits from reducing redundant code. Same programs can be defined within a method in a superclass.

**Interaction between languages** Eiffel can interact with C++ and Java. It uses a set of predefined classes to particularly deal with the process. For instance, JavaClass presents all the classes imported from Java. The latest version can interact with other languages that support .net platform [7]. Examined by the previous principles, this language provides an easy-to-use approach to support language interaction. But Eiffel needs a few additional instructions to support this process rather than reusing the existing ones.

**Easy-to-learn and easy-to-use** In general, Eiffel is an easily learned language. Apart from the use of semicolon, it has similar syntax with other object-oriented languages. The challenge to
learn this language comes from its complication. The powerful language presents a number of novel features. For instance, it has a special exception-handling mechanism. Eiffel is quite easy-to-use. In particular, its unified object model and flexible run-time binding is very helpful for software development [63].

1.2.4 Java

Java [37, 86] was created for embedded devices in the early 1990s. It has a few new ideas. Firstly, Java introduces the web-based application-applet [37], which can directly run on any computer or operating system with a web browser. Secondly, Java presents a build-in security technology-sandbox [35, 74]. It guarantees that web-based applications do not harm local data where applets are used. Thirdly, this language supports platform independence. A Java program does not limit a computer or a platform, but can run on any Java virtual machine (JVM) [59].

Is Java suitable for RAD? We evaluate this language to support RAD from the following aspects.

**Clean concept and no ambiguity** Java has a few easy-to-confused concepts. For instance, Java deals with primitive values and objects [48] individually, and have to design a set of particular classes for those primitive values. A real primitive value is not an instance of these classes, but can be converted to a corresponding object. This approach may cause inconvenience¹, especially when developers use some advanced data structures (eg. Hashtable). In addition, Java uses different binding mechanisms for attributes. Fields are always determined statically, but methods at runtime [37]. The lack of unification may cause confusion in some cases.

**Pure object-orientation** Strictly speaking, Java is not a pure OOPL. Firstly, primitive values cannot be regarded as objects. Their operations are not message-based. Secondly, both predefined and user-defined classes cannot be naturally regarded as objects. To compensate it, Java proposes a mechanism named “reflection” to create references for the classes [48].

¹Java 1.5 overcomes this problem.
**High level programming** Its build-in garbage collector automatically collects spare objects. As the collection occurs unpredictably, this language allows developers to manually trigger it.

**Clear syntax** Java uses most C++ syntax and removes pointer-based operators, therefore presents more readable syntax. But this language still remains a few instructions that may affect program understandability. For instance, a common modifier like `final` may have different semantics for various instructions. A class labelled `final` is "close", and cannot be inherited. A `final` field stands for a constant value that cannot be changed [48].

**Reduction of redundant code** Java reduces code redundancy through dynamically determining methods.

**Interaction between languages** Java is designed to communicate with C++ through the mechanism named Java native interface (JNI). In general, this interaction requires developers to deal with some details [36].

**Easy-to-learn and easy-to-use** Java is an easy-to-learn and easy-to-use language. Its syntax is similar to most the OOPLs, and benefits learners from quickly mastering it. In addition, this small language does not support pointer-based operations. This small and flexible language does not support the complicated features such as multiple inheritance.

### 1.3 Object-based languages

Object-based languages forms the minority in the world of OOPL, but provide more expressivity and flexibility. These languages [82] embed states and behaviors into objects. Multiple objects can share common states and behaviors through "parent" references [90]. Inheritance is represented as a reference between objects.
1.3.1 Self

Self builds on a small number of notions including prototypes, slots and behaviors. Unlike the conventional OOPLs, it effectively represents inheritance and object instantiation through its simple and unified object model. This language does not separately deal with attributes and methods, but unifies them as slots [82].

In detail, this language possesses the following advantages. Firstly, it presents a simple and clear model. Therefore all language features derive from a few key concepts (eg. prototypes and slots). For instance, Self does not need constructors. To create a new object is to clone a prototype object. Secondly, it is more natural to implement singleton classes [90, 31]. Traditionally, class-based languages use some techniques to guarantee a class to produce a common object [28]. Self simplifies this implementation through “parent” references. Finally, Self eliminates infinite regress. Some languages (eg. Smalltalk) regard classes as objects of meta-classes, which are instances of other classes. It leads to an infinite meta-regress [90]. This classless language effectively avoids this problem.

We evaluate its suitability to support RAD development from the following aspects.

**Clean concept and no ambiguity** As one of the simplest OOPL, Self clearly presents language features. It generally regards states and behaviors as slots, and allows developers to access and update them through unified approaches. Through its simple object model, developers easily determine rules to extract slots.

**Pure object-orientation** Self represents every value as an object. All its operations are based on message passing. But it does not provide multilevel access control.

**High level programming** In Self, developers do not need to involve in low-level operations. This language is in charge of memory allocation and release [3, 13].

**Clear syntax** We do not consider this language to define ideal syntax. Self is influenced by Smalltalk.
and reuses most Smalltalk's syntax. The significant differences between Self and other languages affect program readability.

**Reduction of redundant code** Its simple object model provides powerful support for code redundancy. Same or similar code can be reduced through "parent" references.

**Interaction between languages** Self does not present seamless interaction. Firstly, developers have to involve in some details of interaction. For instance, they must create glue functions [3] and specify interaction details like how to convert the data types between these languages. Secondly, Self defines a few primitive operations to particularly support the interaction rather than naturally reuses the existing ones.

**Easy-to-learn and easy-to-use** Self is one of the most easy-to-use languages. The removal of the concept-class makes learners easily understand its object model. However, due to its special syntax, this language is not ideally easy-to-learn, especially to the developers who are familiar with widely used OOPLs.

### 1.3.2 Cecil

Cecil is a pure object-based language, and initially designed to support rapid construction of high-quality and extensive programming [12]. Similar to Smalltalk, this language does not allow the direct access to instance variables, but through automatically generated accessor methods. Cecil allows an instance variable to be updated as a method. It supports parameterized objects and multiple inheritance. Cecil presents a novel mechanism named *predicate object* to dynamically determine the classification of certain objects [14].

We examine its suitability for RAD from the following aspects.

**Clean concept and no ambiguity** Cecil remains several points that may cause confusion. For instance, this language automatically encapsulates instance variables and forbids direct access.
From developers' perspectives, instance variables and methods should be generally considered as attributes and share common access approaches. This language forces the access to instance variables through its automatic-generated methods.

**Pure object-orientation** This language presents all its values as objects, operations as message passing. Cecil also supports multiple inheritance and generic objects.

**High level programming** This language hides all the low-level operations and automatically deals with spare objects [12].

**Clear syntax** This language provides ideal syntax, which is quite similar to most widely used ones. Its additional instructions are clearly defined. For instance, to avoid respectively specifying the interfaces of objects or methods, this language allows a single object declaration to define both an implementation and an abstract interface [12].

**Reduction of redundant code** Like Self, this object-based language possesses powerful expressivity and provides great reusability, and benefits developers from presenting expressive programs and minimizing redundant code.

**Interaction between languages** Cecil can embed C++ code. To extract low-level information like vector indexing and file I/O, this language uses primitive methods. The body of a primitive method consists of a set of pairs constructed by language name and implementation source code. The language name can be either C++ and rti, which stands for the language C++ and the compiler's internal intermediate language respectively [12]. With this approach, developers can call routines written in C++ from Cecil. In general, Cecil presents an easy-to-use way to interact with C++, but still remains a few operations that need developers to manually specify.
Easy-to-learn and easy-to-use  In general, its simple object model and pure object-orientation enables Cecil easy-to-learn and easy-to-use. But Cecil has a complicate algorithm for determining methods to support multiple inheritance. The complication may affect developers to master this language quickly [12].

1.4 Summary

The chapter reviews some influential OOPLs and examines their suitability to support RAD. Firstly, we evaluate class-based languages for RAD. Smalltalk possesses powerful expressivity and flexibility. This pure OOPL hides all low-level operations, but does not provide ideal syntax, which is sharply different with most widely used languages. It may hinder developers to learn and master. C++ is the most unsuitable language for RAD. In order to be compatible with C, this language is not designed as a pure OOPL. Also, to eliminate overhead, C++ does not provide a build-in garbage collector. Eiffel is an excellent language for RAD. It clearly presents concepts and possesses powerful expressivity to reduce code redundancy. But this language proposes so many novel features, therefore affects developers to master it. The simple language-Java has many advantages for RAD development, but have several easy-to-confused mechanisms such as attribute binding.

The object-based languages: Self and Cecil provide extremely simple object models, support pure object-orientation. Due to their expressivity, both of them provide powerful support for reducing redundant code. However, there still exist some unsatisfactory aspects for RAD. For instance, Self reuses most Smalltalk’s syntax, therefore hinders developers to quickly learn it. Cecil requires developers to involve in language interaction. It also presents complicated multiple-inheritance, which may affect developers to learn and use it.
Chapter 2

**FLEXIBO Language Overview**

FLEXIBO is designed as a small, flexible and dynamically typed language. It regards all the first-class values, even classes and methods as objects. The class *Value* is the root class of all the classes. The following diagram presents that all the values including integer and Boolean values derives from that root.

![FLEXIBO's hierarchy](image)

FLEXIBO has some features that do not appear in conventional languages. Firstly, FLEXIBO supports resource control to localize the influence of program errors. It provides two build-in classes: *timer* and *memory* to manage time and memory respectively. Secondly, this language provides an on-line programming design environment. FLEXIBO possesses Unix-like ownership mechanism to guarantee the security and safety of collaboration. Thirdly, FLEXIBO naturally interacts with Java. This language can easily import a Java class, invoke a Java method, even inherit
a Java class or interface. Fourthly, FLEXIBO has many kinds of variables and provides multiple attribute binding ways. Finally, to compensate the lack of static type checking, the language proposes dynamic type constraints to simulate its functionality. Developers are able to redefine default checking rules.

2.1 **FLEXIBO's instructions**

To develop flexible programs, FLEXIBO does not individually deal with expressions and statements, but generally regards the latter as special cases of the former. In general, FLEXIBO has two kinds of expressions: primitive and composite. Primitive expressions consist of values and variables. Composite expressions are composed of the primitive ones.

2.1.1 **Primitive expressions**

Each object contains two components. One is a link to the class that the object belongs to. The other is the record to store data for this object. The class is linked by all its instances, but each object has its own data store. According to FLEXIBO's unified object model, primitive values are also regarded as objects. FLEXIBO automatically interprets primitive values as corresponds objects.

FLEXIBO provides a few kinds of variables such as attribute variables, argument variables, etc. This mechanism—"colorful" variable benefits developers from clearly specifying the expected variables without confusion. A later section particularly discusses these variables.

2.1.2 **Composite Expressions**

Syntax priority is defined for properly evaluating composite expressions. In detail, it is composed of a set of rules to decide expression evaluation order. According to operator priority, the operators with higher priority will be evaluated before those with relatively lower priority. The rule of associativity is used to determine the evaluation order of consecutive operators in an expression.
For instance, the operators with left-associativity are evaluated from left side to right side. The following table presents the priority and associativity of FLEXIBO's operators.

<table>
<thead>
<tr>
<th>Precedence</th>
<th>Operator</th>
<th>Associativity</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>( ), others</td>
<td>—</td>
</tr>
<tr>
<td>1</td>
<td>:</td>
<td>left</td>
</tr>
<tr>
<td>2</td>
<td>[ ]</td>
<td>right</td>
</tr>
<tr>
<td>3</td>
<td>#</td>
<td>right</td>
</tr>
<tr>
<td>4</td>
<td>.</td>
<td>right</td>
</tr>
<tr>
<td>5</td>
<td>&gt;=, =, =, &lt;, &gt;=, &lt;=, !=</td>
<td>right</td>
</tr>
<tr>
<td>6</td>
<td>-&gt;</td>
<td>right</td>
</tr>
<tr>
<td>7</td>
<td>&amp;&amp;, &amp;</td>
<td>right</td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>*, /</td>
<td>left</td>
</tr>
<tr>
<td>10</td>
<td>+, -</td>
<td>left</td>
</tr>
<tr>
<td>11</td>
<td>:=</td>
<td>right</td>
</tr>
<tr>
<td>12</td>
<td>;</td>
<td>right</td>
</tr>
</tbody>
</table>

Table 2.1: Syntax priority

**Composite Expressions**

The following content presents some composite expressions. Their evaluation rules are defined in the above table.

**var** It declares a variable in the environment. This declaration needs a modifier(s) and a variable name. This expression can be either initialized or uninitialized.

**binary choice** It consists of an Boolean expression and two branch expressions. The value of that Boolean expression decides which branch will be executed.

**while** This expression contains a Boolean sub-expression and a body one. The loop doesn't stop until the value of its boolean expression becomes false.

**break** It is used to exit from the innermost enclosing loop. Its evaluation produces a special exception. Only the statement while is able to catch this exception.
continue It terminates the current iteration and start the next one. Similar to break, its execution produces an exception. Only while is able to catch this exception and start the iteration again.

return This expression stops method execution and returns control to callers. It contains an expression. What evaluating the contained expression produces is the result of that method.

try — catch It consists of two expressions and a string to represent catchable error type. If an error is thrown within try and the error thrown matches the formal “type”, the instruction in catch will be evaluated.

assignment This expression is to set values into variables or array elements. Its right expression is evaluated firstly, and then its value is assigned into the left expression.

assert It contains a Boolean expression. If the evaluation of this contained expression produces true, the program continues executing. Otherwise, an exception is thrown to indicate the broken assertion.

block A block constructs a separate environment, which localizes the variables declared within the environment. The evaluation of a block is to evaluate in the contained expression under its own environment.

arithmetic/logical expressions These expressions can be either unary or binary. The former evaluates its contained expression firstly, and then calculates that produced value, returns the final result. The latter evaluates the left expressions firstly, and then the right expressions, finally take a certain operation on these generated values.

We do not present all the composite expressions because of their similarity. The illustration informally explains how to evaluate some instructions and ignore the abnormal cases. A later chapter formally defines the operational semantics for this language.
2.2 Class and Method

FLEXIBO generally regards classes and methods as objects. A normal class can be considered as an "object" of mirror classes. Similarly, mirror methods can instantiate normal methods.

2.2.1 Class

FLEXIBO defines a class as the object that generates other objects. Generally, there exist two kinds of classes in FLEXIBO. One is the predefined classes (e.g., mirror classes). The other is user-defined ones. The keyword class is used to create a user-defined class. The following example shows class declaration.

```plaintext
var cls := class Value (  
    var inclass sa := 1;
    var dynamic da := "dynamic attribute";
    var inclass sm := method s {return s};
);
print cls.sa;
print (new cls).da;
print cls.sm[true];
```

FLEXIBO requires developers to provide an explicit superclass for each declaring class. The class Value denotes FLEXIBO's root class. Every user-defined class inherits it either directly or indirectly. In this case, the class cls directly inherit the root class. Within a class, its attributes can be declared as static or dynamic. The attributes with inclass can be accessed with the class. Access from class instances are not allowed.

FLEXIBO uses different strategies to store these attributes. The values of static attributes are stored in the classes and shared by all its instances. Conversely, the values of dynamic attributes are distributed into each object. The example below shows their differences.
var cls:= class Value ( 
    var inclass sa := 1;
    var dynamic da := "dynamic attribute";
); 

var fo := new cls;
var so := new cls;

print "\n testing static attributes \n";
print cls.sa;
cls.sa := 2;
print cls.sa;

print "\n testing dynamic attributes \n";
print fo.da;
fo.da := "\n modified dynamic attribute \n";
print fo.da;
print so.da;

The following results are printed out:

  testing static attributes
  12
  testing dynamic attributes
dynamic attribute
  modified dynamic attribute
dynamic attribute
After modification, the value of that static variable is updated. The change influences all its objects. On the contrary, the modification in dynamic attributes only happens on individual objects.

FLEXIBO provides several methods to instantiate objects. The following example demonstrates how to generate and initialize objects.

```plaintext
var cls := class Value {
    var sa := 123;
    var dynamic da := "dynamic attribute";
    var dynamic init := method []
        this.da := true;
};
var fol := new cls;
var fo2 := cls.create[];
var fo3 := new cls[]; // initialize an object
print fol.sa;
print fo2.sa;
print fo3.sa;
```

The first approach uses the keyword `new` to produce uninitiated objects from the class. The second one employs the predefined method `create` to create instances. If a class defines the method `init`, it can be used to initialize objects. Developers do not need to explicitly invoke this method, but complete it through the way in the sample program.

FLEXIBO provides the following methods to initialize classes. Firstly, developers can directly assign values into attributes. Secondly, they can declare a block environment for initialization. The following example presents these ways to initialize classes.

```plaintext
var cls := class Value {
    var sa := 123;
    var dynamic da := "dynamic attribute";
    var dynamic init := method []
        this.da := true;
};
var fol := new cls;
var fo2 := cls.create[];
var fo3 := new cls[]; // initialize an object
print fol.sa;
print fo2.sa;
print fo3.sa;
```
var dynamic da;
da := "dynamic attribute";
var inclass sa := new Array[5,5];
{
    var i:=0;
    while (i < 5)
    {
        sa[i,i] := i; i := i + 1 ;
    }
}
print (new els).da;
print cls.sa;

Attributes can be directly initialized through assignments. However, this approach is inconvenient to initialize composite data structure like array. This second approach is particularly useful to simplify array initialization.

FLEXIBO does not separately deal with types and classes. The type of an object is its class. This language provides the predefined method getType to obtain an object’s type.

var cls:= class Value {
    var dynamic da;
    var init := method [] {
        da := "dynamic attribute";
    }
};
print (new cls).getType[];
print (1).getType[];
This example shows how to get the type from an object. For normal objects, their types are the classes where they are declared. To represent the types of primitive values, this language defines a special value wrapping strings. For example, the type of integer is `type {class Integer}`.

Like most object-oriented languages, FLEXIBO provides three modifiers: `public`, `private`, and `protected` to present multilevel access control to attributes.

- **public** As the default modifier, it does not reserve any access restriction.
- **private** The attributes can be accessed under the environment where they are declared.
- **protected** The attributes permit the access from subclasses of the class that declares them.

As illustrated previously, a class is also an object. The class generates other classes is called mirror class. FLEXIBO represents this class with `Class`. Developers can send it a message to generate a class.

```flexib
var Staff := Class.create[Value,quote(var dynamic da:=1)];
var as := new Staff;
print as.da;
as.da:= 2;
print as.da;
```

The predefined method `create` has two arguments. The first one represents the superclass of the producing class. The second one is class body. Note that developers need to use `quote` to avoid evaluating the wrapped program.

### 2.2.2 Method

Similar to classes, FLEXIBO regards methods as objects. A method is an object that consists of two expressions representing arguments and its body respectively, and can be invoked to produce a value.

In general, FLEXIBO has two ways to represent arguments. The first one uses two square bracket to
enclose arguments list. Two arguments are separated by a comma. The second one regards method’s whole argument list as an expression. A well-formed method must have method body. FLEXIBO regards arguments as method’s variables, therefore does not allow directly declaring variable in methods, but blocks.

This language provides two methods to return values to callers. One is to use the statement return to terminate method execution and explicitly return a value. The other approach returns the result that evaluating method body produces.

```plaintext
var cl := class Value {
    var inclass ml := method [x,y] (x||y)
};
print cl.ml[true,false];
```

The case demonstrates the second approach to return a value to the caller without return.

Methods are also first-class values in FLEXIBO. Therefore without invocation, they can be assigned into variables.

```plaintext
var cl := class Value {
    var inclass ml := method [x,y] {
        var temp := x;
        x := y;
        y := temp;
    }
};
var tv := cl.ml;
prompt tv[true,false];
```

In this example, the value of ml(a method) is set into the variable tv, then invoke the value of this variable. The result is same as directly invoking the method.
FLEXIBO uses Method to represent the class to generate methods. A method can be produced through sending a message to this class. The example below demonstrates how to create a method with this approach.

```plaintext
var mtd := Method.create(quote([x,y]),
    quote({var temp := true; return (temp&&x| | y) }));
print mtd[true,false];
print mtd.getType[];
```

The predefined function create is used to produce a method. Its first argument stands for method’s argument list. Note that even if the method has no arguments, this expression is still necessary. The second argument represents method body. The method can be invoked like normal ones.

### 2.3 FLEXIBO’s features

This section exhibits some features of this language. We do not only introduce their implementation, but also present the benefits that these features provide.

#### 2.3.1 Operator overloading

Operator overloading is such a mechanism that simplifies the use of complex types through allowing developers to redefine standard operators. This mechanism provides an approach to reuse the operators. By operator overloading, the use of an object can be simplified as primitive values.

Despite the convenience, this mechanism causes a few problems in many object-oriented languages. For example, C++ allows developers to overload a large number of operators. However, it ignores the fact that multiple overloading may cause inconsistency [85]. For instance, developers intuitionally think the operator >= produces a same value as > and ==. If all the operators can be overloaded, the values of >= may not consist with > and ==. In order to guarantee consistency, they need to overload several operators, therefore lead to complication.
FLEXIBO tries to support operator overloading while minimizes the negative influence above. In general, it needs to solve the following problems. Firstly, how does the interpreter find the methods overloading operators? The methods may exist in a class or its superclasses. Secondly, how does FLEXIBO avoid or minimize confusion while not affect language expressivity?

Table 2.2 shows all the overloadable operators in FLEXIBO. To overload an operator, developers need to declare a variable with its corresponding name and define a method as its value. After compilation, the first operand becomes the host object. The operator is translated as a message, which name is determined in the above table. The second operand is the actual argument. Therefore the evaluation of an expression with an overloaded operator is represented as method invocation. The interpreter searches the method from the host object. If developers do not overload it, an exception is triggered to report that this type does not support this operation. The following example demonstrates how to overload operators.

```plaintext
var c := class Value {  
  var dynamic degree;  
  var add := method s {  
    print "operator overloading\n";
  }
```

<table>
<thead>
<tr>
<th>operator</th>
<th>method's name</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>add</td>
</tr>
<tr>
<td>-</td>
<td>subtract</td>
</tr>
<tr>
<td>*</td>
<td>multiply</td>
</tr>
<tr>
<td>/</td>
<td>divide</td>
</tr>
<tr>
<td>&amp;&amp;</td>
<td>and</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>&gt;</td>
<td>greaterThan</td>
</tr>
<tr>
<td>==</td>
<td>equals</td>
</tr>
<tr>
<td>&lt;</td>
<td>lessThan</td>
</tr>
<tr>
<td>#</td>
<td>invoke</td>
</tr>
<tr>
<td>:</td>
<td>check</td>
</tr>
</tbody>
</table>

Table 2.2: Overloadable Operator
return degree + s;
}

var o := new c;
o.degree := 1;
print o + 2;

In this case, the class c redefines the method add. It will be invoked as long as its object receives the message add. As presented previously, this language possesses two ways to represent arguments. To override operators, the arguments are usually declared as an whole expression. It is convenient to pass the arguments as an object to this redefined method.

Unlike most languages [85], FLEXIBO can determine the operation based on the second operand. Different with the previous overloading approach, it regards the second operand as the host object, the first one as the argument. The following example demonstrates how to apply this approach.

var Even := class Value {
  var dynamic value;
  var multiplied := method s {
    print "operator overloading\n";
    var e := new Even;
    e.value := value * s;
    return e;
  }
}

var e := new Even;
e.value := 2;
print 3 * e;
Clearly, the multiplication that even numbers involve always produces even numbers. In the case, whatever the argument \( s \) is, the produced value is always an even. The special operator overloading is particularly useful to analyze some properties of expressions. A later chapter uses this technique to statically analyze the possible range of an expression.

It is possible that developers overload operators with both the above methods. FLEXIBO checks the left operand firstly. If it properly overloads the operand, FLEXIBO directly passes the second operand as the argument. Otherwise, FLEXIBO checks the second operand with the first operand as the argument. If neither the operands overload operators, an exception thrown indicates misusing.

To minimize the negative influences (eg. inconsistency) of operator overloading, this language takes the following approaches. Firstly, some compound operators are represented by these primitive ones. The value of the operator \( \geq \) is decided by the operators: \( > \) and \( =\). Developers cannot directly overload \( \geq \). Secondly, it provides two kinds of equalities. \( == \) stands for pointer equality. Its value is decided physically. Developers cannot overload this operator. The other equality \( =\) compares values by default and is allowed to be overloaded.

FLEXIBO does not completely eliminate inconsistency. For instance, it has these overloadable operators: \( >, \geq, <\). Any two of them can represent the third operator. Overloading all these operators leads to inconsistency. However, FLEXIBO still remains those operators. It is a trade-off for simplicity. Although it is possibly to use fewer operators, it may cause program complication, therefore reduce understandability.

2.3.2 Exception

"An exception is a run-time event that may cause a routine call to fail." [63] During the development of a program, there may exist some cases where developers do not have the certainty that a piece of the code is going to work properly. Exception handling is used to ensure that the programming can execute correctly. Even if some exception happens, it can be controlled in a scope.
Exception handling is an important feature for object-oriented programming languages. Both C++ [76] and Java [71] use the keywords: try, catch and throw to support this mechanism. Eiffel [93] has a special approach to handle exception. The structure rescue is composed of some instructions to change the state and at most one special instruction retry. When an exception is triggered, Eiffel stops executing method body, invokes the structure rescue. As the special statement retry is invoked, that method body is executed once more. If there is not the instruction rescue, the language returns the control to the caller and reports an exception [63].

Most these languages have the following limitations:

- Developers have to specify many exceptions that may be triggered.
- They provide a little support for capturing unpredicted exceptions.

To present an easy-to-use and powerful exception-handling mechanism, FLEXIBO need to handle the following challenges. Firstly, as a dynamic typing language, how does this language identify each kind of exceptions? Traditionally, object-oriented languages identify exceptions with data types. Secondly, how does this language catch a set of exceptions, and avoids repeating each individual case. Many languages forces developers to identify all exception that may be triggered.

Similar to C++ and Java, FLEXIBO uses the structure try – catch to capture exceptions. In addition, the keyword thisException is used to identify the recently thrown exception. Unlike those languages, FLEXIBO uses strings to represent exceptions. As an exception is thrown, FLEXIBO checks whether the string of that triggered exception matches the string following catch. If yes, the statements defined within the structure catch are executed. Otherwise, FLEXIBO returns the exception to the caller. The string of the actual exception does not need to be exactly same as the one following catch. If the latter is a substring of the former, this exception can be captured. This matching way can be used to catch a set of exceptions like Java-related exceptions.

```flexibolo
var cls := class Value {
    var inclass mth := method [] {
```
try {
    var jc := Jclass "java.lang.Integ" ;
    return new jc;
}
catch "java" {
    print thisException;
}

print cls.mth[;];

This program tries to declare a non-existed Java class, and triggers an exception to indicate this error. The structure catch can capture any Java-related exception. Without this feature, developers have to code many times for each case. An empty string is a substring of any string. If developers present an empty string following catch, all the catchable exceptions are captured. This mechanism also enables FLEXIBO to catch unpredicted exceptions.

2.3.3 Ownership and control

FLEXIBO provides an on-line programming environment, which allows users to develop programs, implements other developers’ specifications and tests other programs. To guarantee secure and safe collaboration, FLEXIBO presents an easy-to-use way to defence unauthorized operations.

FLEXIBO adopts an Unix-like ownership mechanism. It allocates each user an owner. Users cannot change their own owners. This language manages the relations between owners. A developer can take any operation on his own data. For example, they are also able to add, modify and remove relations between other owners and themselves.

Compared with Unix system [72], this ownership mechanism is more complicated. Unix system provides three levels of relations for users. It enables different developers to have various operation
permissions. By comparison, the relations between FLEXIBO's developers can be more complex. For instance, it can be the relation between designer and tester, contractor and implementer, etc.

FLEXIBO uses the keywords: owned and own to declare an ownership expression and an ownership value. The keyword ThisOwner denotes the owner of the current user. Users can obtain the owner of an expression through the predefined method getOwner if it is under ownership control.

In general, FLEXIBO has three kinds of relations between owners. The most general relation is strange. If an owner holds this relation with another, it presents that he regards another owner as a stranger. self represents the relation between developers and themselves. Others are user-defined. Developers are able to present multilevel relations for different owners. Each relation corresponds to a relation ID. Table 2.3 shows those relations and their corresponding ID. The relation number ranges from 0 to 255. The ID 0 is reserved for strange, 1 for self. The predefined method setRelation is provided to modify the relations between owners. This method needs two arguments. The first argument is the other owner's ID, the second argument is a relation number.

<table>
<thead>
<tr>
<th>Relation</th>
<th>Relation ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>strange</td>
<td>0</td>
</tr>
<tr>
<td>self</td>
<td>1</td>
</tr>
<tr>
<td>user-defined</td>
<td>2-255</td>
</tr>
</tbody>
</table>

Table 2.3: Owner Relation

The relations between owners may not be symmetric. Diagram 2.2 shows this property. The left
user specifies that he holds the relation 11 with the right user and defines the operation permission of this relation as 10. The relation and permission manage the incoming access to his controlled data. In contrast, the right user presents different relation and operation permission between them.

Similar to Unix system, FLEXIBO allows developers to modify operation permission for a relation. For example, an owner may change it to allow other owners to read data. This language represents operation permission as an integer. Each binary bit of that integer manages an operation. Table 2.4 presents these operations and their corresponding binary representation. There exist dependency relations between some operations. For instance, if an inheritable class must be readable.

<table>
<thead>
<tr>
<th>Operation</th>
<th>Binary Rep</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read</td>
<td>1</td>
</tr>
<tr>
<td>Write</td>
<td>10</td>
</tr>
<tr>
<td>Execution</td>
<td>100</td>
</tr>
<tr>
<td>Creation</td>
<td>1001</td>
</tr>
<tr>
<td>Inheritance</td>
<td>10001</td>
</tr>
<tr>
<td>Permission change</td>
<td>100010</td>
</tr>
<tr>
<td>Implementation</td>
<td>1000010</td>
</tr>
<tr>
<td>Variable dec</td>
<td>10000010</td>
</tr>
<tr>
<td>Role exchange</td>
<td>111111111</td>
</tr>
</tbody>
</table>

Table 2.4: Operation Representation

Figure 2.3 presents ownership controlled expressions and the process how to complete ownership check. According to the diagram, each of these expressions contains two sub-expressions. One
is to check incoming access. The other is the protected expression. If the access succeeds in passing
ownership checking, this operation is passed into the contained expression. Note that the protected ex­
pression does not take any further operations to validate the access. Its behavior is same as a normal
expression. Of course, it is also possible that the contained expression is ownership controlled.

**FLEXIBO** provides the predefined method `setPermission` to modify operation permission.
It also has two arguments. The first one represents the relation. The second one is for the new
operation permission. The following example presents how to use these predefined methods to
change ownership relation and access permission.

```java
var owned ht := new (Jclass "java.util.Stack");
var kk := ((raw ht).getOwner[]);
raw ht.setRelation[kk,1];
raw ht.setPermission[1,5];
home := 1;
print home;
```

This example shows that the developer sets the relation between himself and the one whose owner
ID is 11 as 1, and then changes the permission as executable and readable represented by 5. Note
that the keyword `raw` enables developers to operate on the expressions rather than their values. In
the case, the access control is based on the expression `ht`.

### 2.3.4 Language interaction

Language interaction is the ability for a language to interact with other languages. This mechanism
is widely used in many object-oriented languages (eg. Java [57]). However, most of these languages
do not present an easy-to-use interaction approach.

This language is able to naturally interact with Java. It defines shallow structures for the inter-
action. For example, **FLEXIBO** defines Java Class, Java Interface, Java Method corresponding to
Java's structures. Developers can easily import Java's classes, create Java's objects, invoke Java's methods and even inherit classes or interfaces from Java.

2.3.5 Colorful Variable

**FLEXIBO** systematically analyzes various kinds of variables, and categorizes them as local variable, argument variable, attribute variable, cast variable and defined environment variable, therefore allows developers explicitly to indicate the *color* of variables.

Local variables are the temporary variables declared in blocks. **FLEXIBO** looks for these variables from the innermost block. The following example demonstrates how to determine a local variable.

```plaintext
var Person := class Value (  
  var age := 18;  
  var type := "Person";  
  var inclass printing := method [age] {  
    var age := 20;  
    print loc age;  
  }  
);  
Person.printing[19];
```

Since the "color" of the variable age is clearly specified, the value 20 is printed out.

Argument variables are the variables that are declared as method's arguments. If the variable labeled by `arg` does not appear in method's argument list, an exception is thrown to indicate that the expected variable does not exist.

```plaintext
var Person := class Value (  
  var age := 18;  
  var age := 18;
```
var type := "Person";
var inclass printing := method [age] {
    var age := 20;
    print arg age;
};

Person.printing[19];

The number 19 is printed out. Argument variables are allowed to share common names with local variables. The keyword arg explicitly indicates that the following variable is an argument.

For the variables that do not specify the "color"s, this language determines them from innermost environments.

var Person := class Value {
    var age := 18;
    var type := "Person";
    var inclass printing := method [] {
        print type;
    };
};

Person.printing[];

This case outputs the string Person. FLEXIBO searches the variable from the block, to the method printing, then to the class Person.

In general, there exist three "color"s of variables to represent different attributes: cast variable, attribute variable and defined variable. The next subsection discusses these variables individually.
2.3.6 Flexible Attribute Binding

Code redundancy is a serious problem that affects software quality. Reuse is regarded as an effective way to reduce redundant code. But this problem still exists in many object-oriented programming languages more or less. The following example (coded in Java) shows how attribute binding causes code redundancy.

```java
public class Person {
    int age;
    static String type = "Person";
    public Person() {
    }
    public void print() {
        System.out.println("Person is printing "+type);
    }
}

public class Staff extends Person {
    static String type = "Staff";
    public Staff(int age) {
        this.age=age;
    }
    public void print() {
        System.out.println("Staff is printing "+type);
    }
    public static void main(String[] s) {
        Person ps = new Staff(23);
        ps.print();
    }
}
System.out.println("printing an attribute "+ps.type);
}
}

Java uses a complicated mechanism to determine (bind) attributes. It decides fields at compilation, but methods at runtime. In the case, developers need to repeat the method print for several times to bind the proper attributes type. Otherwise, it cannot bind the types declared in the subclasses.

FLEXIBO tries to apply language expressivity and flexibility to minimize code redundancy. FLEXIBO provides a flexible binding approach. Unlike Java, it does not separately handle fields and methods, but decides both of them at runtime by default. With FLEXIBO, the example above can be simplified as follows.

```
var Person := class Value (  
  var dynamic age := 18;  
  var type := "Person";  
  var printing := method [] {  
    print this.type;  
    print " is printing ";  
    print this.age;  
  };  
);  
var Staff := class Person (  
  var type := "Staff";  
);  
var lecturer := new Staff;  
lecturer.printing[];
```

The duplicate programs can be removed through reusing printing defined in the superclass Person.
As a pure object-oriented programming language, FLEXIBO generally handles fields and methods. With the keywords like `att`, `def` and binding expression, it supports different binding approaches for attributes. The following content aims to use FLEXIBO to develop practical programs, therefore shows the advantages of this language.

Firstly, we launch the case with attribute variables. The following example presents three classes: `Person`, `Staff` and `Lecturer`. Clearly, there exists inheriting relations among them. Each of those classes has an attribute-type, which represents the type of this class. It is no doubt that all the classes have individual types. For instance, the type of `Person` is the string “Person”. Its subclass `Staff` has more specified type “Staff”. Similarly, the type of `Lecturer` is presented as “Lecturer”.

```flexib
var Person := class Value {
    var static type := "Person";
    var printing := method [] {
        print "I am a ",
        print att type;
    };
};

var Staff := class Person {
    var static type := "Staff";
};

var Lecturer := class Staff {
    var static type := "Lecturer";
};

(new Person).printing[];
(new Staff).printing[];
(new Lecturer).printing[];
```
In the method `printing`, we use attribute variable `type`. This attribute always traces the running environment. In this case, these three objects have their own `type`. Developers do not need to repetitively define this method. With attribute variables, the method can automatically bind the proper one for each object.

Clearly, the benefit comes from reducing code redundancy. With this kind of attributes, developers do not need to repeat similar code in each subclass. For complicated hierarchy, this mechanism is greatly helpful to simplify programs.

The second case makes a demonstration for `def` variables. We reuse those classes shown in the first case. The method `printing` is changed to print a Boolean value to check whether an object is a "Person". Clearly, for all the instances of these classes, the answer is expected to be Yes.

```plaintext
var Person := class Value {
    var static isPerson := true;
    var printing := method [] {
        if ((def isPerson) == true)
            print "Yes, I am a person";
        else
            print "No, I am not";
    }
};
var Staff := class Person {
};
var Lecturer := class Staff {
    // var static isPerson := false;
};
var lecturer := new Lecturer;
lecturer.printing[];
```
Any subclass of Person should keep unchanged output. In the other words, this method is "closed". Further inheritance will not modify the original result. However, this requirement may not be met in some cases. If developers use att isPerson rather than the one shown in this case, the returned result may be different. Assume that the class Lecturer has an extra attribute named isPerson, att isPerson represents this newly declared attribute instead of the expected one. def variables are always bound to some fixed environments (eg. declaration).

The final case shows the usage of cast variables. In some cases, developers need to access to the attributes declared in the superclasses. But these attributes may not be directly accessed because of overriding. cast variables provide the support for specifying binding environments for attributes.

```
var Person := class Value {
    var static type := "Person";
};
var Staff := class Person {
    var static type := "Staff";
};
var Lecturer := class Staff {
    var static type := "Lecturer";
    var printing := method [] {
        print "I am a ";
        print att type;
        print ", also a ";
        print cost type@Person;
        print " and a ";
        print cost type@Staff;
    };
};
```
var lecturer := new Lecturer;
lecturer.printing[];

In this case, the method printing tries to print all the types. Since the attributes are overridden, the
class Lecturer cannot use previous methods to access to the ones in its superclasses. To explicitly
show the types defined in Person and Staff, we need to use def attributes to explicitly access
to the attributes. It benefits developers from presenting the specified attributes, particularly the
overridden ones.

2.3.7 Type Constraint

FLEXIBO does not support static type checking, but supports dynamical type constraints. The
following example demonstrates how to apply this technique to represent type constraint.

var Person := class Value (  
    var type := "Person";
);  
var Rick:Person;
Rick := new Person;
print (true):Person;

This expression contains two sub-expressions. The first one represents a restricted expression, the
second one stands for its "type", which defines the rule how to check the first expression.

This language proposes several default checking principles. The restricted type of an usual
object should be identical to its class, or its superclass. FLEXIBO predefines the values like Int
and Bool to denote the types of primitive values. These types(also values) reserves straightforward
rules. The following example gives an demonstration to check the type for integer.

var Person := class Value (  
    var type := "Person";
);
var inclass printing := method[name, age:Int]
    (print name; print "'s age is "; print age;)
);

Person.printing["Rick", 45];
Person.printing["Rick", "45"];

Program execution stops at the final line. An exception will be thrown to indicate type inconsistency.

More interestingly, developers are allowed to override the default rules, and present stronger or weaker type checking. The following example defines a stronger checking rule. It requires that the restricted type of an object must be identical to its class.

var Staff := class Value
    var type := "Staff";
    var inclass check := method[stf]
        (thisclass===(stf.getType[[]]))
    );

var rick:Staff;
rick := new Staff[];
rick := ture;

This mechanism is helpful to flexibly present type constraint. A later chapter will apply the technique to implement code generation for a specification program.

2.4 Summary

This chapter introduces the language-FLEXIBO, which is designed to support prototype development. As a pure object-oriented language, it regards all its values as objects. Primitive values, classes and methods can be represented through its unified object model. This language does not individually deal with expressions and statements, but consider the latter as special cases of the
formal. This consideration simplifies programming instructions and enables developers to present flexible programs. FLEXIBO supports operator overloading, but minimize its traditional negative influences (e.g. inconsistency). It presents a novel exception-handling mechanism, which is able to catch both a set of exceptions and unpredicted exceptions. FLEXIBO provides an on-line programming environment and proposes a Unix-like ownership mechanism to guarantee secure and safe collaboration. This language also seamlessly interacts with Java. It allows developers to import Java class, declare Java object, and inherit Java class and interface. In addition, it provides many kinds of variables (e.g. attribute variable), and has multiple binding approaches for reducing redundant code. To compensate the lack of static type checking, FLEXIBO checks type constraints at runtime and allow developers to override the default checking rules.

There still exists some features that this experimental language does not support. Firstly, FLEXIBO does not support generic programming. This mechanism enhances reusability and extendability through type parameterization. Generic programming is widely used in C++ programs through template [8]. Eiffel, C# and the latest Java provides more flexible support to this mechanism. Secondly, FLEXIBO does not support the iterations such as for and do-while, the conditional instruction case. These instructions can be added into FLEXIBO through the similar way for others. Thirdly, FLEXIBO does not allow the invocation from Java. Language interaction enables FLEXIBO easily to use the properties of Java. In some cases, Java also need to reuse FLEXIBO programs. The current implementation does not give the invocation interface to Java. Fourthly, the current implementation and its illustration is based on java 1.4. It does not check whether FLEXIBO is able to smoothly integrate with the latest version of Java. Fifthly, by now FLEXIBO does not provide local design environment. The initial design goal of FLEXIBO is to give an on-line developing environment. In fact, if this language provides a local console, it would be convenient for developers to quickly get familiar. The change to the current implementation only happens on the place where to load source code. Local environment stores source code into files. The code of online environment is retrieved from browsers.
Chapter 3

Object-oriented Language Interaction

This chapter investigates the interaction between FLEXIBO and other object-oriented languages. It shows the interaction strategies for not only the basic objects and primitive values, but also some advanced features such as cross-language inheritance.

3.1 Introduction

Language interaction is widely used in many languages including C++ [85], Java [37] and Eiffel [63]. In general, it helps developers under the following situations:

- The current language does not include some features, but other languages do.
- Developers want to access to a library or application that has been already created in other languages, therefore extends reusability from a single language to multiple ones.
- Access to low-level features. Some high-level languages(particularly interpreted) do not support the access to low-level resources[57].

Many object-oriented programming languages [57, 61] regard language interaction as a fundamental mechanism. However, due to heterogeneous structures of different languages, the interaction may lead to complications, and reduce program readability. For instance, JRuby [84] can interact with Java. It requires a number of instructions to particularly handle the process. Secondly, their
different syntax forces developers to use the grammars of multiple languages. Thirdly, developers may need to handle the data returned from target languages [12, 34]. Some languages even require developers to manage interaction detail.

We propose an approach for seamless language interaction. Operating on target languages is almost same as on local languages. By comparison, this approach requires languages to take more responsibilities of the process. For instance, it needs to automatically translate data types between the local languages and target ones. Furthermore, it provides cross-language inheritance for reusability. In the world of target languages, inheriting relation is still reserved. In the other words, the object of a subclass is recognizable in the inherited language. Its type in target languages is generally regarded as the inherited class.

In detail, there exists the following technical challenges to import classes, create objects and invoke methods from target languages. Firstly, how does the local language handle the data from target languages? Secondly, how does the language explicitly determine a method from a group sharing a common name?

We present the solutions as follows. Firstly, the local language declares a special kind of classes and objects denoting the directly imported classes from target languages. Each of them contains a reference to a native class in target languages. The latter stands for the objects of the former. Similarly, they have references to native objects. For both of them, the contained values, as agents are in charge of all the operations in the world of target languages. Secondly, the approach requires two mapping tables to manage data translation between the local language and target languages. These tables clearly specify all data types and their corresponding types. A relatively challenging problem is that multiple methods may share common names if target languages allows method overloading. To solve the problem, we try to extract the information from arguments. Both argument length and its type array are essential to explicitly determine a method at runtime.

The most difficult question is how to inherit a cross-language class or interface. We expect that the inheritance allows the objects of subclasses to be used as the instances of its superclass in
target languages. In addition, how do we allow target languages to invoke the methods in the local language?

For the advanced features, we propose the solutions as follows. We present a special kind of classes, which derive from cross-language classes either directly or indirectly. Similarly, a kind of objects are proposed to correspond to the classes. Each of the special objects has a reference to a cross-language object, which is generated by the cross-language class of its linked class. These objects and classes use the wrapped values to interact with target languages. Since the contained values can represent inheriting relations, for the special objects and classes, those relations are still reserved in the world of target languages.

The previous method is not applicative for inheriting interfaces because they cannot instantiate objects. We introduce a kind of virtual objects. The interface will generate a dynamic class according to the running context. The instance of the generated class will be assigned as the cross-language object of a virtual object.

This chapter does not only launch the strategies for interacting FLEXIBO with other languages, but also exemplify the process between FLEXIBO and Java. Technically, we meet the following challenges.

- How does Java handle the data from FLEXIBO? FLEXIBO regards everything as an object. But Java has two kinds of data types: primitive type and reference type. The reference data is an object, but the primitive values are not.

- How does FLEXIBO deal with the data from Java? As presented above, Java has objects and primitive values. We expect that its primitive values corresponds to FLEXIBO's primitive ones.

- How does FLEXIBO obtain a Java method? There is a conflict between Java and FLEXIBO. Java supports method overloading. Multiple methods may share a common name. However, each name corresponds to a unique method in FLEXIBO.
• How does Java invoke a method in FLEXIBO? Java’s interface contains a set of fields and unimplemented methods. When a class inherits an interface, it needs to complete all the unimplemented methods declared in that interface. However, Java cannot directly invoke those implementation because FLEXIBO’s class is not a class in Java.

The following sections apply the strategies presented previously to meet these practical problems.

3.2 Class, Object and Method

This section studies the fundamental part of language interaction. In detail, it firstly deals with basic instructions including object and class, and then proposes the technique to invoke methods of target languages. For further discussion, $L$ and $T$ are used to represent the local language and the target language respectively.

3.2.1 Interaction Strategy

Before investigating the interaction, we study how to import a class from target languages and create its objects. An extra keyword is essential to identify this cross-language import. Since every class has its own path, the import needs to specify an accurate “location”. The declaration of a cross-language object needs to invoke an explicit constructor in target languages. Many object-oriented programming languages support constructor overloading. Therefore it is possible that many constructors share a common name. To explicitly invoke one of them, both class name and the additional information about arguments are indispensable. This process is almost the same as method invocation. It will be generally discussed later on.

Interaction of Class and Object

In object-oriented paradigm, object is the most fundamental concept. To deal with language interaction, we firstly present the interaction about objects. The following problems need to be solved.
• What is the type of an object of \( L \) in \( T \)? Obviously, \( T \) is not able to recognize its type in \( L \).

• If an object is generated from a cross-language class, what type is it in the local language \( L \)?

• How can we access to its attributes from the language \( L \)?

In response to the challenges, we propose the following solutions. Many object-oriented programming languages [34, 37, 61] have single-hierarchy structure. There are root classes in those languages. For instance, Smalltalk and Java provide the class \texttt{Object} as their root class. For the objects generated from cross-language classes, their contained native objects presents their types in \( T \). On the contrary, for the ones instantiated by normal classes in \( C \), the root class like \texttt{Object} is assumed as their types in \( T \). To meet the second challenge, the language \( L \) presents a special kinds of objects, which represent the ones declared from \( T \). Each of these objects contains a reference to an native object(see Figure 3.1) generated from \( T \). So there exists various kinds of objects, each of them has an individual approach to access to its attributes. In this case, to access to a cross-language attribute is to look up it from the refereed object. If it is a valid attribute, further operations occur on the object(or its class). Otherwise, an exception is thrown to indicate that the attribute does not exist.

To clearly present the differences between the local properties and the imported ones, we define the term(cross-language class) as the class in \( L \) that contains a reference to link to a native class in \( T \). Similarly, a cross-language object is defined as the object in \( L \) that contains a reference to link to an object of \( T \). Cross-language classes can instantiate cross-language objects. According to the definitions, both cross-language classes and cross-language objects are the properties in \( L \). Unlike the normal ones, both of them have cross-language references. The following diagram clearly presents the relations between these notions.
Interaction of Primitive Values

We also need to deal with the communication for primitive values like integers and Boolean values. Similar to objects, their translation must make the values in a local language recognizable in target ones. For instance, if an integer in $L$ is pass to $T$, it is expected to be interpreted as an integer in target languages. We define the tables (see Figure 3.2) to manage the translation for the sent and received data between $L$ and $T$. As the local language sends a value to $T$, a table interprets it to a proper value. Similarly, a value returned from $T$ is translated to a correspond one in $L$ according to the other table.
Interaction of Methods

To invoke a method in $T$ has the following challenges. The first one is how to explicitly obtain that method. Due to method overloading, method name cannot be the unique identification to a method. It is possible that many methods share a common name, but have different arguments. The second challenge is how to pass the arguments from $L$ to $T$ and invoke that method.

As illustrated above, only method name is inadequate to identify a method. We try to extract additional information from arguments. The number of arguments and the array constructed with argument types aid to determine a method from a group of methods with a common name. To invoke this method needs the support from the target language. If $T$ can regard a method as a high-level object, we can send a predefined method to the method with the local object and arguments. For the languages supporting delegation [89] or function pointer [85], we are able to use these mechanisms to dynamically invoke a method.

3.2.2 Interaction between FLEXIBO and Java

We exemplify the process with FLEXIBO and Java. The current language-FLEXIBO wants to use some utilities in the target language-Java.

FLEXIBO uses the keyword Jclass followed by a path to import a Java class. This language provides several approaches to instantiate a Java object. The example below demonstrates how to obtain a Java class and create its instances.

```plaintext
// test java interface
print Jclass "java.awt.event.ActionListener";

// test java class and objects
var jc := Jclass "java.net.Socket";
var jpk := Jclass "java.lang";
print jpk.Integer;
```
According to the example, developers are able to import both normal classes and interfaces with that keyword. Java regards the latter as a special class without method implementation. In addition, the keyword can be reused to import a Java package. It is helpful to frequently use classes in a common package. FLEXIBO provides a number of approaches to create a Java object. The first one is to invoke FLEXIBO-style constructor. It produces an uninitialized Java object. It is same as to invoke a java constructor with no arguments. The second approach executes the predefined method create to obtain an object.

![Class hierarchy](image)

**Figure 3.3: Class hierarchy**

The diagram(Figure 3.3) presents the relations between FLEXIBO and Java. To integrate the properties of Java, FLEXIBO defines a set of notions(eg. Java class), which are transparent to developers. All Java instances are wrapped as object of Java object. Similarly, Java's classes correspond to Java class. An instance of Java method may contain a group of Java's methods sharing a common name. With this model, we can embed Java’s properties into FLEXIBO. Classes,
objects and methods directly map to corresponding instructions in FLEXIBO.

Interaction of Object and Primitive Values

FLEXIBO defines Table 3.1 for mapping from FLEXIBO's types to their corresponding ones in Java. These mapping rules are quite easy-to-understand. Any primitive types such as Integer and Float directly map to its corresponding primitive types in Java. Since Java separately deals with objects and primitive values, we use \( p \) and \( r \) to indicate primitive type and reference type respectively. The agent objects mainly include cross-language objects or classes. Their contained native objects or classes will be the agents to communicate with Java. Excluding these primitive types and cross-language objects, other data types generally correspond to Java's root class \( \text{Object} \). This process is completely transparent to developers, who do not need to take any responsibility.

The translation is two-way. FLEXIBO cannot recognize the raw data returned from Java. It needs similar rules to interpret the data received. Table 3.2 defines type mapping rules from Java to FLEXIBO. It presents that all Java's primitive types is directly mapped to FLEXIBO's primitive types. Java's reference type \( \text{String} \) maps to FLEXIBO's primitive type \( \text{String} \). Slightly different with the previous table, this one presents a special kind of objects, each of which wraps a value. The translation(see Figure 3.4) unpacks this kind of objects and obtains the wrapped value. This mechanism is particularly useful to use Java's utility library(eg. Vector and Hashtable). The wrapped

<table>
<thead>
<tr>
<th>FLEXIBO</th>
<th>Java</th>
</tr>
</thead>
<tbody>
<tr>
<td>Integer</td>
<td>( int_p )</td>
</tr>
<tr>
<td>Float</td>
<td>( float_p )</td>
</tr>
<tr>
<td>Long</td>
<td>( long_p )</td>
</tr>
<tr>
<td>Double</td>
<td>( double_p )</td>
</tr>
<tr>
<td>Boolean</td>
<td>( Boolean_p )</td>
</tr>
<tr>
<td>Character</td>
<td>( char_p )</td>
</tr>
<tr>
<td>String</td>
<td>( String_p )</td>
</tr>
<tr>
<td>Agent obj</td>
<td>agent</td>
</tr>
<tr>
<td>Other</td>
<td>( Object_p )</td>
</tr>
</tbody>
</table>

Table 3.1: FLEXIBO-Java Mapping Table
Table 3.2: Java-FLEXIBO Mapping Table

<table>
<thead>
<tr>
<th>Java</th>
<th>FLEXIBO</th>
</tr>
</thead>
<tbody>
<tr>
<td>int_p</td>
<td>Integer</td>
</tr>
<tr>
<td>float_p</td>
<td>Float</td>
</tr>
<tr>
<td>long_p</td>
<td>Long</td>
</tr>
<tr>
<td>double_p</td>
<td>Double</td>
</tr>
<tr>
<td>Boolean_p</td>
<td>Boolean</td>
</tr>
<tr>
<td>char_p</td>
<td>Character</td>
</tr>
<tr>
<td>String_p</td>
<td>String</td>
</tr>
<tr>
<td>wrapped object</td>
<td>original type</td>
</tr>
<tr>
<td>Other object</td>
<td>Java Object</td>
</tr>
</tbody>
</table>

expressions benefit developers from easily using these classes.

Wrapped objects are also helpful to reserve the properties (eg. values and operations) of FLEXIBO's objects. For instance, as an object sent to Java is returned, its original properties in FLEXIBO is expected to be recovered. The following example demonstrates this.

```java
var jc := Jclass "java.util.Hashtable";
var fc := class Value {
    var fmethod := method [s] {
        print s;
    };
};
```
var temp := new jc;
var tv := wrap 0;
temp.put [tv, wrap (new fc)];
(temp.get [tv]).fmethod ["printing"];

The expression wrap 0 prevents 0 from being interpreted as an integer in Java. For the expression wrap (new fc), the wrapping operation enables the returned object to be recovered as the original one. Without the wrapping process, the returned value is a raw Java object, therefore cannot receive the message fmethod.

Interaction of Method

Java allows method overloading. Therefore a method name may correspond to several methods. FLEXIBO supports high-order functions. Methods can be represented as normal objects. The following example demonstrates how to obtain a group of Java method according to a method name.

var jint := Jclass "Java.lang.Integer";
print jint.toString;

This small program generates the result as follows.

JavaMethod (public Java.lang.String Java.lang.Integer.toString ()
public static Java.lang.String Java.lang.Integer.toString (int)
public static Java.lang.String Java.lang.Integer.toString (int, int)
)

In this example, there exist three methods sharing the name toString.

To explicitly gain an method from Java, FLEXIBO needs method name, argument length and the array of argument types. This determination occurs before method invocation. With the additional information, the method can be decided explicitly.
In this case, the interpreter can select the second method from all the possible ones and invoke it, and then produces the value 1 as the result.

Some Java methods may have complicated argument structures. For instance, the type of its argument may be composite data structures like array. To identify an explicit method, the interpreter needs to check array length, dimension amount and its component type. All of them are essential ingredients to determine a method.

### 3.3 Cross-language Inheritance

The previous section shows the fundamental aspects of interaction. It does not only introduce the interaction strategies, but also demonstrates that process through Java and FLEXIBO. However, it lacks the support for the advanced features like cross-language inheritance. This section describes how to inherit a class or an interface from target languages, and exemplifies the technical details through the interaction between FLEXIBO and Java.

#### 3.3.1 Cross-language Inheritance from a class

Inheritance has an important role in object-oriented paradigm. It provides reusability, and is closely related to polymorphism. Through the interaction of two object-oriented programming languages, cross-language inheritance benefits developers from more powerful reusability.

To seamlessly inherit a cross-language class, we have the following challenges.

- Does object instantiation of a subclass produce an object of its superclass?

- As an object of a subclass is sent to target languages, how does this object reserve its inheriting relation in $T$?
• How do we access to the attributes declared by its superclass?

In the local language \( L \), there exists two kinds of classes, which are subclasses of cross-language classes. One is the class that directly inherits some class in \( T \). The other is the one which superclasses inherit from a class(cross-language) in \( T \). We generally discuss these cases. Unlike the normal classes in \( L \), each of them has an extra attribute to link to the cross-language class inherited either directly or indirectly (see Figure 3.5).

Similarly, we also use a special kind of objects to represent the ones that are generated from the classes above. In addition to the normal properties in common objects, they have extra attributes to denote cross-language produced by cross-language classes. Figure 3.5 presents the relations between these notions and the previous ones.

![Figure 3.5: Class inheritance](image)

With this model above, we are able to meet previous challenges. As a class (inheriting class in Figure 3.5) inherits a cross-language one either directly or indirectly, the cross-language class is stored into this class to identify its special superclass. When this class instantiates an object, this contained class also produce a cross-language object. This cross-language object is stored into that object to be returned (inheriting object in Figure 3.5). Therefore the object returned includes an additional cross-language object.
Unlike a normal object in $L$, as this special object is sent to $T$, it extracts the contained cross-language object as its delegation to communicate with the target language $T$. Since this contained object is recognizable by $T$ because it contains a native object, the inheriting relation of the original object can be reserved in $T$. Similarly, the classes inheriting from $T$ also have this property. When they communicate with the target languages, the contained class takes the delegation role. The types of these objects in $T$ can be obtained through the contained cross-language ones.

Since both these classes and objects above contain agents to communicate with $T$, the access to their attributes can be completed through the agents. For instance, when we attempt to assign a value to an attribute of an object, the value is passed to its cross-language object. This agent is in charge of any further operations. If it is a dynamic attribute, the agent object can be directly updated (through its contained native object). On the contrary, if the attribute belongs to a class, the modification message is sent to the class (cross-language) that generates the agent object.

### 3.3.2 Cross-language inheritance from an interface

A few languages provide a special kind of classes, each of which contains a set of fields and unimplemented methods. For instance, Java presents the classes named “interface”. They cannot instantiate objects, but be inherited. This subsection studies how to inherit these special classes. Compared with the inheritance from normal classes, to inherit from interfaces is much more challenging. In general, there are the following tasks to present natural interaction for interfaces:

- How does a class in $L$ inherit an interface from $T$, and produce a local subclass?
- How do we generally deal with object creation for the classes inheriting interfaces and the normal ones?

The first task can be completed through the similar approach for normal classes. $L$ regards the classes inheriting interfaces (cross-language) as a special kind. Each of them has a particular attribute to represent a cross-language class, which natively contains an interface of $T$ (see Figure 3.6).
This interface, as a delegation, is in charge of the communication with $T$.

![Diagram of interface inheritance]

By comparison, the second task is much more challenging. Firstly, it is impossible for interfaces to create instances. Secondly, the target language may need to obtain the implementation of a certain method. For instance, Java commonly uses the mechanism-event handling to deal with windows programming. Any event like window opening corresponds to a fixed-name method. When the event is triggered, Java virtual machine (JVM)\cite{59} automatically invokes that method. The challenge exists in how to invoke these methods implemented with $L$ in the world $T$. This process is similar to cross-language method invocation, but in a reverse direction.

Since interfaces cannot produce instances, we introduce a special kind of cross-language objects. Different with the normal ones, they are not really generated by certain classes (or interfaces), but as delegations to store some auxiliary information. These objects contain two attributes: a native object and a cross-language interface. Virtually, this cross-language interface "generates" the cross-language object. Initially, the contained native object is null because native interfaces cannot instantiate objects, but will be updated by dynamically produced class according to the running environment. The diagram (Figure 3.6) presents the relations between these notions.
To invoke the method in $L$ from $T$, we need to use some fundamental properties of languages. As $T$ tries to invoke a fix-name method, the local language looks up the virtual object (see Figure 3.6). This virtual object contains a dynamically produced native object, which class (also dynamically generated) presents all the behaviors about methods. Many object-oriented languages like Java support mirror classes or reflection. They enable developers to regard methods as objects, and then sends some predefined messages to execute the methods. For the languages like C++ [85], we can use function pointers to execute it.

Thus completing the second task needs the following steps. Firstly, according to the interface, a virtual object is “generated”. So the object of the class inheriting that interface contains this virtual object. Secondly, when this object is sent to $T$, a dynamic class will be produced. Since an interface cannot generate objects, the dynamic class is in charge of this creation. Its instance will be set into the virtual object as an agent to communicate with the target language. If the target language needs the implementations of some methods, they are able to obtain them through the dynamically generated object.

To inherit cross-language interfaces is really challenging. We assume that the target language can dynamically produce classes. This requirement can be met in a few languages supporting the mechanism-interface. For instance, Java allows developers to use Proxy to dynamically generate a class, therefore creates its object.

### 3.3.3 Interaction between FLEXIBO and Java

The previous subsection exhibits the strategies for cross-language inheritance. Now we use FLEXIBO and Java to demonstrate the interaction in practice.

#### Class inheritance from Java

FLEXIBO allows its classes to inherit from Java. To produce seamless interaction, this language does not require additional instructions for the process. Cross-language inheritance is exactly same as
the normal inheritance. The following example presents how to inherit a Java class in this language.

```javascript
var jc := Jclass "java.util.Hashtable";
var fc := class jc {
    var fmethod := method s {
        print (new super[]).toString[];
    };
}

var temp := new fc;
temp.fmethod[];
print temp.toString[];

var tv := wrap 0;
temp.put[tv, "hello"];  // Note that it is compulsory to use the keyword wrap to wrap the integer 0,
print temp.get[tv];     // therefore avoid it to be interpreted as a primitive value in Java.
```

The following result is printed out.

```text
{}{}hello
```

This example demonstrates how to inherit a Java class, access to an attribute and invoke methods in its superclass. Firstly, the program invokes a FLEXIBO method `fmethod`. It generates an object from its superclass, therefore obtains an empty Hashtable, and then invokes the method `toString` on this object. The process produces the same result as directly invoking this method. After inserted an element, the Hashtable has its first record. The Java method `get` returns the element for the declared table.
FLEXIBO applies the strategies illustrated previously to interact with Java. It presents a special kind of class named ClassWithJava to represent the classes inheriting from Java. Each of them contains a reference to its inherited Java class (cross-language in Figure 3.5). A special kind of objects CompJavaObj are generated from these special classes. Each of these objects has a Java object (cross-language in Figure 3.5). When the classes produce instances, they firstly require the contained Java class (cross-language in Figure 3.5) to generate a Java object, and then yields and returns an object (CompJavaObj) embedded with that Java object. As an agent of the object (CompJavaObj), its Java object (cross-language) manages all Java-related operations. For instance, its Java type is what the native object contained in the Java object (see Figure 3.1) is. Any update to Java attributes or method invocation on Java methods happens on the Java object (actually its native object).

Figure 3.7: Cross-language class inheritance

Figure 3.7 presents the relations between these notions. As a special class, ClassWithJava stores an additional attribute to represent a Java class (cross-language in Figure 3.5). Correspondingly, CompJavaObj stands for the objects instantiated from those special classes. Each of them contains an extra attribute for a Java object (cross-language in Figure 3.5). Java init object is particularly designed for inheriting the Java classes with multiple constructors, but no default ones (no arguments). Java does not allow these classes to be inherited. To settle this problem, we present
Java *init* object. As a class inheriting one of the classes instantiates objects, the contained Java class(cross-language in Figure 3.5) generates a Java *init* object rather than Java *object*. It will be converted as a Java *object* according to the running context.

```javascript
var jc := Jclass "java.lang.Integer";
var fc := class jc {
    var init := method [s] {
        print s;
    };
};

var obj := new fc[1];
print obj.toString[];
```

In the case, the evaluation of `new fc[1]` needs two steps. Firstly, `new fc` generates an instance for `fc`. Simultaneously, the superclass `java.lang.Integer` generates an object. Since the superclass has no default constructor, the instantiation of its subclass cannot directly produce a Java *object*, but Java *init* object. Secondly, according to the arguments of `fc`, the latter is converted as the former, which contains a Java native object (`new Integer(1)`).

With the approach, these objects can be recognized in Java. The translation reserves their corresponding types(through agent objects), therefore provides more detailed information for method invocation or other applications.

**Interface inheritance from Java**

FLEXIBO also allows inheriting a Java interface, which consists of a set of methods expected to be implemented. The following example presents a small event-based program.

```javascript
var jc := Jclass "java.awt.event.ActionListener";
```
var fc := class jc {
    var actionPerformed := method s {
        (Jclass "java.lang.System").exit[0]
    }
};

var temp := new fc[];

var button := new (Jclass "java.awt.Button");
print button.setSize[100,100];
button.addActionListener[temp];

var frame := new (Jclass "java.awt.Frame");
frame.setSize[200,200];
frame.add[button];
frame.show[]

This example shows how to inherit a Java interface. As a special kind of Java classes, Java interface
can be imported through the same approach as Java class. Since the method actionPerformed
is unimplemented, the subclass fc presents its implementation. An instance can be produced from
this subclass. After the declaration of a Java object, this event object is planted into that object. As
this event is activated, the program in the method actionPerformed will be executed.

The interaction is implemented through the approach presented previously. Each of these
classes inheriting from Java interfaces constructs a special FLEXIBO class, which contains an extra
attribute-Java interface(cross-language in Figure 3.5). A particular kind of objects are defined as
virtual objects of Java interfaces. Each of these objects contains a reference to a native object in
Java and a reference to an Java interface(cross-language in Figure 3.5). The former one is an object
generated dynamically, which stores Java-related data. The latter one presents its Java type. As
this object is sent to Java, that language identifies the type of the object through its included Java interface and communicates with Java by its contained Java object.

Figure 3.8 presents a part of class hierarchy, which is closely related to cross-language interface inheritance. As a subclass of Java class, Java interface has a reference links to a native Java interface. Any derived class stores this superclass(cross-language in Figure 3.6). Correspondingly, Java interface object represents the “instance” of Java interface. Unlike a normal Java object, it has an additional attribute-Java interface for its “creation”.

![Cross-language interface inheritance](image)

We also study how Java obtains the implementation of a fix-named method from FLEXIBO. When Java receives FLEXIBO’s object, it firstly generates a dynamic Java class and produces its native object. Then, this native object is set into that virtual object(Java interface object in Figure 3.8) as an agent to interact with Java. To implement this, we use the technology named Proxy [6] provided by Java. It allows dynamically producing a Java class and instantiating it according to the running context. When JVM tries to obtain the implementation of a method, it looks up this generated object. Since this object includes the information about implementing a certain method, the concrete implementation can be obtained from there. FLEXIBO allows developers to regard methods as normal objects and provides the predefined method invoke to execute them. The generated value will be the final result of the method with the fixed name.
3.4 Related work

C++ [85] builds on the procedure language C [51], and inherits all the properties that the latter provides. Strictly speaking, we cannot accurately call it as interaction, but extension. It is also able to interact with assembly language. This mechanism is proposed for more efficient execution for pieces of code. However, this interaction requires a special instruction [28] to support. The interaction reduces programming readability, causes inconvenience in programming development and debugging and forces developers to master the grammars of both the languages.

Java can interact with C++ through a particular mechanism named Java native interface (JNI) [57]. In general, writing native methods for Java programs needs to the following steps. The first one is to generate a Java class that declares a native method. The Java class includes the declaration for that method and a main method to call that native method. The next step is to compile the Java class containing the methods. Thirdly, a header file is generated for the native method. Fourthly, developers implement the native method in C++ [57]. Before running this Java program, they must compile the header and implementation files into a shared library file. Clearly, this process needs developers’ involvement and requests developers to be familiar with the syntax of both Java and C++.

Eiffel interacts with both Java and C++. This language provides a number of predefined classes, each of which corresponds to a kind of instructions in Java. For instance, the class JAVA.OBJECT stands for Java objects, JAVA.CLASS for Java classes, JAVA.ARGS for Java arguments. It also defines mapping tables for translating primitive values. Although this language is able to interact with Java easily, developers must involve some details like explicit declaring of Java objects [7] by JAVA.OBJECT. Eiffel also interacts with C++ with similar restrictions. The latest Eiffel [7] supports .net cross-language interaction, which enables this language to communicate with other ones support this platform [87].
It is noticeable that some languages such as Jython [75] and JRuby [84] interact with Java. However, there still exist a few limitations to hinder natural interaction. Jython requires an additional keyword self to compulsorily appear as the first formal argument. It will reduce readability in some cases. For instance, as a class inherits a Java interface, it tries to complete unimplemented methods. Intuitively, these methods keep identical signatures with their declaration. Due to this special requirement, this subclass has to insert self in front of all formal arguments. JRuby uses a set of keywords to require developers to specify some structures like Java methods. For example, developers need to code as follows to obtain a Java method.

```java
string_class = Java::JavaClass.for_name("java.lang.String")
string_class.method("valueOf", "int")
```

The string "valueOf" represents method name. "int" is a real argument. Clearly, it is inconsistent with the normal methods declared in JRuby.

### 3.5 Summary

The chapter begins with introducing how to translate the primitive values and objects from the local language to target languages. With a mapping table, each kind of data maps to a corresponding kind of data in target languages. Since the interaction is two-day, another mapping table manages to translate the data returned from target languages. For the classes or objects(cross-language or inheriting), this approach can reserve their specified types in the world of target languages. Beside the fundamental translation for primitive values and objects, this chapter also presents how to determine and invoke cross-language methods. More interesting, it shows the ways to inherit classes or interfaces in target languages, therefore allows developers to use cross-language utilities and enhance reusability.
In general, it proposes a solution for language interaction between FLEXIBO and other object-oriented languages. It technically includes the features like the translation of primitive values, cross-language inheritance, etc. We exemplify the technique through the interaction between FLEXIBO and Java. This approach provides more natural interaction. Cross-language features can be used in a similar way as the corresponding features of the local language. It requires few additional instructions for the interaction. Therefore we claim that the interaction implemented with this technique provides more “friendly” cross-language integration. Although the illustration is based on FLEXIBO, we believe that these strategies are also helpful for the interaction of other object-oriented languages.
Chapter 4

Generating BSPlib-C code from LOGS Specifications(I)

This chapter shows a tool to automatically translate a concrete form of specifications to C code linked with BSPlib. A LOGS specification for Bulk-Synchronous Parallelism is a relation of an initial state, a final state and some intermediate states. Nondeterminism and parallelism respectively correspond to disjunction and conjunction. Advanced specification commands build on the fundamental ones. Before generating target code, the translator checks syntax, freedom of communication interference, type consistency and communication dependencies. Static analysis (including both static checking and translation) is presented through abstract interpretation. It is shown that a few laws are complete for transforming any specification into a normal form. These laws are satisfied by the abstract functions. We demonstrate the actual effects of the abstract functions by applying them on the normal form.

4.1 Introduction

Bulk-Synchronous Parallelism [91] is a programming paradigm based on variable sharing and global synchronization. It decouples communication and synchronization for parallel computation [41]. A BSP program is composed of a number of supersteps, each of which has three ordered
steps: local computation, communication between processes and synchronization. Local computation commands appear between consecutive synchronization. Most communications occur till the next synchronization point. These points separate the execution of a BSP program as a sequence of supersteps.

A BSPlib-C program is a normal C program linked with BSPlib [40] that supports several function calls. Command \texttt{bsp.pushregister} registers a piece of memory to be shared for communications; command \texttt{bsp.popregister} releases a piece of memory from registration; command \texttt{bsp.sync} synchronizes with other processes’ \texttt{bsp.sync} commands; command \texttt{bsp.put} sends some data from a local address to an address on a remote process, and the communication is delivered at the following synchronization; command \texttt{bsp.get} requests some data from an address on a remote process to a local address, and the data arrives at the following synchronization.

Chen and Sanders [19] presents an intermediate specification language LOGS to formally support MIMD program development. It makes explicitly the intermediate global states at synchronization points. Communications are abstracted in LOGS. For a vector \( w \) of program variables, the primitives of LOGS are commands on \( w \) taking \( n \) steps. The following table [19] lists the primitive commands of LOGS.

<table>
<thead>
<tr>
<th>( p )</th>
<th>n-step command</th>
</tr>
</thead>
<tbody>
<tr>
<td>( P \oplus Q )</td>
<td>sequential composition</td>
</tr>
<tr>
<td>( P \smallfrown Q )</td>
<td>nondeterministic choice (disjunction)</td>
</tr>
<tr>
<td>( P \sqcup Q )</td>
<td>parallel composition (conjunction)</td>
</tr>
<tr>
<td>( \phi f )</td>
<td>recursion</td>
</tr>
</tbody>
</table>

An \( n \)-step command [19] is represented as \( (p)_n \) where \( p = p(w, w_0, \ldots, w_{n-1}, w) \). In it each \( w_k \) with \( k < n \) denotes the state at the \((k+1)\)-th intermediate synchronization point. For example, \( (x + 1 = x_0 = \overline{x} - 1)_1 \) is a 1LOGS command in which the program variable \( x \) is increased by 1 before its first intermediate synchronization point \( x_0 \) and increased by 1 again by
termination. Another example of 0-step command $\langle \frac{x^2}{2^2} \leq 4 \rangle_0$ represents a local computation without synchronization, and the final state of $x$ is implicitly related to its initial state through an inequation. The sequential composition of two processes merges into a longer one:

$$\langle p(\overline{w}, w_0, \cdots, w_{n-1}, \overline{w}) \rangle_n \triangleright \langle q(\overline{w}, w_0, \cdots, w_{m-1}, \overline{w}) \rangle_m$$

$$= \langle \exists w \cdot (p(\overline{w}, w_0, \cdots, w_{n-1}, w) \land q(w, w_n, \cdots, w_{n+m-1}, \overline{w})) \rangle_{n+m}$$

where the final state of the first process is associated with the initial state of the second process, and then the interface is hidden [18]. No additional synchronization point is inserted by sequential composition. This reflects the fact that, in BSP, the sequential composition can be placed either at a synchronization point or between two consecutive synchronization points. The nondeterministic choice between two nLOGS commands is the disjunction of their internal predicates. The parallel composition of two nLOGS commands is the conjunction of their internal predicates. More useful commands can be derived from the primitive ones like binary conditional, loop, etc.

Top-down design of a program starts from an abstract formal specification, then refine the specification to a more concrete form that incorporates design decisions reflecting a commitment to algorithmic and data representation, and can be further implemented with executable code. The refinement of specifications corresponds to removal of nondeterminism. LOGS has been applied to both numeric applications with data parallelism and distributed computing with task parallelism.

A specification is concrete [18], if it is composed of only sequential and parallel compositions and a finite number of 0-step commands, each explicitly expressed as $\langle \overline{w} = f(\overline{w}) \rangle_0$, and 1-step commands, each expressed as $\langle w = g(\overline{w}) \land \overline{w} = h(\overline{w}, w) \rangle_1$ where $f, g$ and $h$ are expressions. The refinement from abstract specifications to concrete ones needs decision makings and is normally done manually.

A concrete specification can be transformed into program code. Chen and Sanders [18] studied refinement from LOGS to a simplified BSP language and introduced two commands of variable protection. Communication commands are added into specification with the refinement laws, and communication interference can be detected with algebraic laws of variable protection. The method
is mainly suitable for manual calculation.

This chapter intends to transform concrete LOGS specifications directly into MIMD programs with sub-programs in parallel, each located on one process. That means the length of the target C code grows linearly with the number of processes. This allows the translator to calculate the approximated values of some expressions (e.g. indices of array access) for specific individual processes during the phase of static analysis and hence results in faster and safer code.

Before generating target code, the translator needs a few steps to check the syntax, freedom of communication interference, type consistency and communication dependencies between processes. The validation is based on abstract interpretation [20, 21]. For instance, code generation becomes an abstract function that transforms a program into a string in C. In general, compilation can be considered as a special translation whose target language is a low-level one.

Communication interference commonly exists in shared-memory parallelism. It is implemented as runtime exception in BSPlib [40]. For basic LOGS specifications accurately, it is possible to check communication interference. Advanced commands such as multiple parallel composition and for-loop iteration (appear in the next chapter) may require abstraction and approximation.

Typical type systems are founded in proof theory and defined with inference rules in the style of operational semantics. Simple (stateless) type inference rules directly correspond to abstract functions on types, which can be regarded as abstract values [23]. If type inference rules depend on the context (i.e. the state of the type checker), it is still possible to encode the context as an argument of an abstract function.

Normal form is a widely used technique to check whether the definition of an abstract function is appropriate. Under some algebraic laws, the syntax of a language may collapse to a normal form. This is known as the completeness of the laws with respect to the normal form in algebraic semantics. This chapter attempts to demonstrate the effect of an abstract function by applying it to the normal form of LOGS specifications and calculate the result. This suffices to show the effect of the function on every specification, if every specification can be reduced to the normal form, and
the abstract function satisfies the algebraic laws.

1-step LOGS commands represent stepwise design of BSP programs with synchronization, while 0-step commands is a specific characteristic of BSP's local computation without synchronization. In general, there exist two kinds of special 1-step commands:

\[
\begin{align*}
[p(w, w)] &= (q \land w = w)_1 \quad \text{early transition} \\
[p(w, \bar{w})] &= (q \land \bar{w} = w)_1 \quad \text{late transition}.
\end{align*}
\]

An early transition (implementable with `bsp_put()`) is a 1-step command that may change state before the synchronization point but maintains a stable state between the intermediate and final states. For instance, the specification

\[
[\bar{x} = \overline{y} + 1] \cup [\overline{y} = \bar{x} - 1]
\]

is a parallel composition of two early transitions. The values of \(x\) and \(y\) are changed at the synchronization point. The new values remain unchanged in the final state. Most numeric computations with data parallelism can be characterized with early transitions. A late transition (implementable with `bsp_get()`) keeps a stable state up to the synchronization point but may have a different final state from the intermediate state. Other processes can access a process's initial state by observing its first intermediate state at the synchronization point. This is particularly convenient to task-parallel computations such as the dining-philosopher problem [18].

A concrete 1-step specification \(\langle x = g(\bar{x}) \land \bar{x} = h(\bar{x}, x) \rangle_1\) can always be transformed into the sequential composition of an early transition and a 0-step command with a fresh temporary variable \(y\):

\[
[\bar{x} = g(\overline{y}) \land \overline{y} = \overline{y}] \; \cup \; [\bar{x} = h(\bar{x}, \overline{y})]_0,
\]

or similarly, the sequential composition of a 0-step command and a late transition. Here only deals with early transitions. The inconsistency in the original concrete specification can be detected automatically, subject to a certain degree of abstraction (e.g. the abstract interval analysis of array indices).
The LOGS translator is implemented in the untyped Object-Oriented language-FLEXIBO [17]. It is able to simulate the behaviors of types, allows user-defined types (as objects) and checks type consistency at runtime. Program constructs of the language can be inherited and extended for translation from a given source language (e.g. LOGS) into a more efficient target language (e.g. C/C++). Runtime checking (e.g. type checking) performed by the FLEXIBO program actually become static analysis for the source language.

Each translator is implemented as a reflection system consisting of the classes that override FLEXIBO's own classes of program constructs. For example, FLEXIBO's if-then-else statements are objects of a class called SemBinaryConditional. Pre-defined methods such as evaluation and printing can be overridden in the classes extending SemBinaryConditional. A FLEXIBO program can be converted into a kind of reflected program under a given reflection system. When the new reflected program is evaluated, user-defined evaluation method instead of the pre-defined method will be invoked. FLEXIBO provides a platform on which various static-analysis methods can be systematically developed in an Object-Oriented manner.

Context-free static analysis is defined as an abstract function on individual program constructs and is implemented as a (polymorphic) method with no side effect in FLEXIBO. Features of existing reflection systems can be inherited by new static-analysis tools through FLEXIBO's inheritance mechanism. Context-dependent static analysis, however, is often defined with inference rules in the style of operational semantics, and directly corresponds to a (polymorphic) FLEXIBO method with side effects. Therefore FLEXIBO has integrated both styles of semantic models in a single framework of implementation.

4.2 Syntactical checking

FLEXIBO pursues extreme expressivity and flexibility, thus removes many syntactical restrictions in conventional languages [85, 37]. For instance, the operator $F_1 \# F_2$ represents a method invocation
of \( F_1 \) with argument \( F_2 \). \( F_1 \) and \( F_2 \) can be arbitrary expressions. Even an expression like 1 \# 2 is syntactically correct, although its evaluation would generate a runtime exception, since the integer 1 cannot provide the service of a method.

If we ignore syntactical restrictions for priority order and parenthesis, FLEXIBO's syntax is completely flat:

\[
F ::= F \parallel F \mid F ; F \mid \text{if } F \text{ then } F \text{ else } F \mid \text{early } F \mid F \otimes F \mid \text{after } F = F \mid F + F \mid F \land F \mid \neg F \mid \text{before } F \mid x \mid v.
\]

The program operators are, in order, the parallel composition \( \parallel \) (MIMD parallelism), sequential composition \( ; \), binary conditional \text{if}-\text{then}-\text{else}, early transition, logical \text{and} \( \otimes \) between internal predicates of early transitions, internal predicate in which \text{after } F \text{ stands for the final state of an individual program variable } F, \text{arithmetic plus } +, \text{boolean and } \&, \text{boolean negation } \neg, \text{initial state before } F \text{ of a variable } F, \text{program variable } x \in \mathcal{X} \text{ and constant value } v \in \mathcal{V}. \text{Note that we have listed only the basic program constructs used by LOGS. The operators } +, \& \text{ and } \neg \text{ are merely representatives of arithmetic and logical operators allowed in LOGS. The specification (4.1) can be written in the above syntax as follows:}

\[
P_{ex} \equiv \text{early (after } x = \text{before } y + 1) \parallel \text{early (after } y = \text{before } x + 2). \quad (4.2)
\]

Unlike FLEXIBO, the specification language LOGS has the following hierarchical syntax:

\[
P ::= P \parallel P \mid S
\]

\[
S ::= S ; S \mid \text{if } S \text{ then } S \text{ else } S \mid S \land S \mid E \Rightarrow S \mid T
\]

\[
T ::= \text{early } I
\]

\[
I ::= I \otimes I \mid W
\]

\[
W ::= A = E
\]

\[
A ::= \text{after } X
\]

\[
E ::= E + E \mid E \land E \mid \neg E \mid \text{before } X \mid X \mid V
\]

\[
X ::= x
\]

\[
V ::= v.
\]
Note that there exist two additional operators: nondeterministic choice $\sqcap$ and conditional magic $\triangleright S$. They are by-products of static analysis and only appear in the normal form of specifications (see section 5.3). For example, an if-then-else command can be decomposed as the nondeterministic choice between two exclusive conditional magics [43].

Each of the following notions represents a set of LOGS's instructions.

<table>
<thead>
<tr>
<th>$P$</th>
<th>LOGS processes</th>
</tr>
</thead>
<tbody>
<tr>
<td>$S$</td>
<td>LOGS sequential processes, $S \subseteq P$</td>
</tr>
<tr>
<td>$T$</td>
<td>LOGS early expression, $T \subseteq S$</td>
</tr>
<tr>
<td>$I$</td>
<td>LOGS internal expression</td>
</tr>
<tr>
<td>$A$</td>
<td>LOGS after expression</td>
</tr>
<tr>
<td>$E$</td>
<td>LOGS normal expression</td>
</tr>
</tbody>
</table>

As a convention, we use $P, P_1, P_2, \cdots$ to denote individual specifications in $P$, and let $S, S_1, S_2, \cdots$ denote individual sequential processes in $S$ and so on. We also use $S_0$ to denote the set of sequential processes without the two additional operators.

The abstract boolean functions formalize the above syntactical restrictions.

$$\alpha, \alpha_S, \alpha_I, \alpha_E, \alpha_X : \mathcal{F} \to \{ \text{true}, \text{false} \}$$

A FLEXIBO expression $F$ is a well-formed LOGS specification if and only if $\alpha(F) = \text{true}$.

**Def 1**

$$\alpha(F_1 \parallel F_2) = \alpha(F_1) \land \alpha(F_2) \quad \alpha(F_1 ; F_2) = \alpha_S(F_1 ; F_2)$$

$$\alpha(\text{if } F_1 \text{ then } F_2 \text{ else } F_3) = \alpha_S(\text{if } F_1 \text{ then } F_2 \text{ else } F_3)$$

$$\alpha(\text{early } F) = \alpha_S(\text{early } F)$$

**Def 2**

$$\alpha_S(\text{if } F_1 \text{ then } F_2 \text{ else } F_3) = \alpha_E(F_1) \land \alpha_S(F_2) \land \alpha_S(F_3)$$

$$\alpha_S(F_1 ; F_2) = \alpha_S(F_1) \land \alpha_S(F_2) \quad \alpha_S(\text{early } F) = \alpha_I(F)$$

**Def 3**

$$\alpha_I(F_1 \otimes F_2) = \alpha_I(F_1) \land \alpha_I(F_2) \quad \alpha_I(F_1 = F_2) = \alpha_W(F_1 = F_2)$$
Def 4 \[ \alpha_W(F_1 = F_2) = \alpha_A(F_1) \land \alpha_E(F_2) \]

Def 5 \[ \alpha_A(\text{after } F) = \alpha_X(F) \]

Def 6 \[ \alpha_E(F_1 + F_2) = \alpha_E(F_1 \land F_2) = \alpha_E(F_1) \land \alpha_E(F_2) \]

\[ \alpha_E(\neg F) = \alpha_E(F) \]

\[ \alpha_E(\text{before } F) = \alpha_X(F) \]

\[ \alpha_E(x) = \alpha_E(v) = \text{true} \]

Def 7 \[ \alpha_X(x) = \text{true} \]

We assume that, by default, every boolean function returns \textit{false} for any expression undefined in the above rules. It becomes trivial to check that \( \alpha(P_{ex}) = \text{true} \).

The above syntactical restrictions directly correspond to a few polymorphic methods without side effect in \textsc{Flexibo}:

```javascript
var LOGS := class Reflection (  
  var SemExp := class (superclass.SemExp) (  
    var alphaP := method [] false;  
    var alphaS := method [] false;  
    var alphaW := method [] false;  
    var alphaE := method [] false;  
    var alphaA := method [] false;  
    var alphaX := method [] false;  
  );  
  var SemOpOr := class (superclass.SemOpOr) (  
    var alphaP := method []  
      e1.alphaP[] && e2.alphaP[];  
  );
```
var SemSeqComp := class (superclass.SemSeqComp) {
    var alphaP := method [] alphas[];
    var alphas := method [] el.alphas[] && e2.alphas[];
};

var SemBinaryCondition := class (superclass.SemBinaryCondition) {
    var alphaP := method [] alphas[];
    var alphas := method [] b.alphaE[] && el.alphaS[] && e2.alphaS[];
};

var SemLogsEarly := class (superclass.SemLogsEarly) {
    var alphaP := method [] alphas[];
    var alphas := method [] e.alphaI[];
};

var SemOpAnd := class (superclass.SemOpAnd) {
    var alphaI := method [] el.alphaW[] && e2.alphaI[];
    var SemOpEqualTo := class (superclass.SemOpEqualTo) {
        var alphaI := method [] alphaW[];
        var alphaW := method [] el.alphaA[] && e2.alphaE[];
    };
    var SemOpAfter := class (superclass.SemOpAfter) {
        var alphaA := method [] e.alphaX[];
    };
    var SemBinOperator := class (superclass.SemBinOperator) {
        var alphaE := method [] el.alphaE[] && e2.alphaE[];
    };
    var SemUnaryOperator := class (superclass.SemUnaryOperator) {
        var alphaE := method [] e.alphaE[];
    };
    var SemOpBefore := class (superclass.SemOpBefore) {
        var alphaE := method [] e.alphaX[];
    };

The above FLEXIBO program provides a real example how static analysis is implemented in the language. Reflection is the root reflection system, a class containing internal classes that represent program constructs. By extending Reflection, LOGS also becomes a reflection system whose internal classes extend the original classes for program constructs. The method invocation LOGS.flexibo[] adjusts the reflection system through multiple inheritance so that, for example, the features of SemBinOperator can be inherited by classes such as SemOpAdd and SemOpLogicalAnd. The reflection system LOGS can then be used as a template to convert syntactical constructs. Methods like alphaP will be re-directed to the corresponding classes in LOGS through dynamic binding.

4.3 Basic assumptions, normal form and completeness

Static analysis methods based on abstract interpretation are defined recursively for every program constructs. We shall use the technique of normal form to demonstrate that the abstract functions are indeed properly defined. The following laws are assumed to be true for basic LOGS specifications.

Law 1 (Basic assumptions)
(1) **associativity of** \((\cdot \parallel \cdot), (\cdot ; \cdot), (\cdot \cap \cdot) \text{ and } (\cdot \otimes \cdot)\)

(2) **distributivity of** \((\cdot ; \cdot) \text{ into } (\cdot \cap \cdot)\)

(3) \(\text{if } E \text{ then } S_1 \text{ else } S_2 = (E \triangleright S_1) \cap (\neg E \triangleright S_2)\)

(4) \(E \triangleright (S_1 \cap S_2) = (E \triangleright S_1) \cap (E \triangleright S_2)\)

(5) \(E \triangleright (S_1 ; S_2) = (E \triangleright S_1) ; S_2\)

(6) \(E_1 \triangleright (E_2 \triangleright S) = (E_1 \land E_2) \triangleright S\)

(7) \(\text{true} \triangleright S = S\).

The above list is not a complete list of all laws satisfied by the concrete semantics of LOGS specifications. For example, parallel composition also has commutativity in the concrete semantics of dynamic behavior. Nevertheless the list is complete for the static-analysis methods in this chapter.

Under the above assumed laws, LOGS syntax collapses to the following normal form where

\[\prod_{j=1}^n K_j \equiv K_1 ; K_2 ; \cdots ; K_n\] and \(\bigotimes_{i=1}^m A_i \equiv A_1 \otimes A_2 \otimes \cdots \otimes A_m\).

**Norm 1**

\[P ::= \prod_{l=1}^s \prod_{k=1}^n S_{kl}\]

\[S ::= \prod_{j=1}^n K_j\]

\[K ::= E \triangleright T\]

\[T ::= \text{early } \bigotimes_{i=1}^m \text{ after } x_i = E_i\]

The above normal form can be merged into one line where parallel compositions choices are located in the outmost layer, and then nondeterministic, sequential compositions, conditionals and finally, early transitions. The collapse of the syntax stops at the level of LOGS expressions, which are not further reducible:

\[\prod_{l=1}^t \prod_{k=1}^s \prod_{j=1}^n E_{jkl} \triangleright \left( \text{early } \bigotimes_{i=1}^m \text{ after } x_{ijkl} = E_{ijkl} \right)\]

where \(t\) is a constant, but \(s = s(t), n = n(s, t)\) and \(m = m(n, s, t)\) are dependent functions.
Theorem 1 (Completeness of basic assumptions) Any well-formed basic LOGS specification can be reduced to the above normal form under the laws of basic assumptions.

Proof. We prove by induction on the structure of program constructs.

1. Need to show that, if a program construct $F$ satisfies $\alpha_I(F) = \text{true}$, then it can be reduced to the normal form $\bigotimes_{i=1}^{m} \text{after } x_i = E_i$:

   (a) indeed, if $F = (\text{after } F_1 = F_2)$, and hence $\alpha_{X}(F_1) = \text{true}$ and $\alpha_E(F_2) = \text{true}$, then $F = \bigotimes_{i=1}^{1} \text{after } x_i = E_i$ where $x_1 = F_1$ and $E_1 = F_2$;

   (b) otherwise, if $F = F_1 \otimes F_2$ where $\alpha_I(F_1) = \alpha_I(F_2) = \text{true}$, then by induction assumption, we have $F_1 = \bigotimes_{i=1}^{m} \text{after } x_i = E_i$ and $F_2 = \bigotimes_{i'=1}^{m'} \text{after } x_{i'} = E_{i'}$, and thus $F = \bigotimes_{i''=1}^{m+m'} \text{after } x_{i''} = E_{i''}$ where for any $i'' \leq m$, $x_{i''} = x_{i''}$ and $E_{i''} = E_{i''}$, and for any $i'' > m$, $x_{i''} = x_{i''-m}$ and $E_{i''} = E_{i''-m}$.

2. Need to present that, if a program construct $F$ satisfies $\alpha_S(F) = \text{true}$, then it can be reduced to the normal form $F = \prod_{k=1}^{s} \prod_{j=1}^{n} K_{jk}$:

   (a) if $F = (\text{early } F_1)$, and hence $\alpha_I(F_1) = \text{true}$. We have $F_1 = \bigotimes_{i=1}^{m} \text{after } X_i = E_i$, hence $F = T$ and $T = (\text{true } \triangleright T)$. Therefore we have $F = \prod_{k=1}^{s} \prod_{j=1}^{n} (\text{true } \triangleright T) = \prod_{k=1}^{1} \prod_{j=1}^{1} K_{jk}$

   (b) if $F = (\text{if } F_1 \text{ then } F_2 \text{ else } F_3)$, thus according to the basic assumptions, we have $F = (E \triangleright S_1) \cap (\neg E \triangleright S_2)$. $\alpha_E(E) = \alpha_S(S_1) = \alpha_S(S_2) = \text{true}$, by induction assumption, let $E$ be an expression $E \in E$ and $S_1, S_2$ be the expressions as follows:

   $S_1 = \prod_{k=1}^{s} \prod_{j=1}^{n} K_{jk}$

   $S_2 = \prod_{k'=1}^{s'} \prod_{j'=1}^{n'} K_{j'k'}$

   So we have $(E \triangleright S_1) \cap (\neg E \triangleright S_2)$ by the law(4) in Law 1.
by the law (5) in Law 1

by definition

by the law (6) in Law 1

by the associativity of \((\cdot \cdot)\)

by the associativity of \((\cdot \cap \cdot)\)

If \(j = 1\), \(K_{jk}' = (E \cdot E_{1k}) \cdot T_{1k}\). Otherwise for any \(j > 1\), \(K_{jk}' = K_{jk}\). Similarly, if \(j' = 1\), \(K_{jk}' = (\cdot \cdot E_{1k'}) \cdot T_{1k'}\). Otherwise for any \(j' > 1\), \(K_{jk}' = K_{jk'}\).

For any \(k'' \leq s\), \(K_{jk''}'' = K_{jk''}''\) and \(n'' = n\), and for any \(k'' > s\), \(K_{jk''}'' = K_{jk''}''(k'' - s)\) and \(n'' = n'\).

(c) otherwise, if \(F = (F_1 \cdot F_2)\), then \(\alpha_S(F_1) = true = \alpha_S(F_2)\). So by induction assumption, we have \(F_1 = \cap_{k=1}^n K_{jk}\) and \(F_2 = \cap_{k'=1}^{n'} K_{jk'}\). Therefore we have \(F = F_1 \cdot F_2\) and \(p_1 = (k'' \mod s')\) and \(p_2 = (k' \mod s)\). Thus, if \(j \leq n\), \(K_{j''k''}'' = K_{j''p_1}''\). Otherwise, for any \(j > n\), \(K_{j''k''}'' = K_{j''p_1}'' K_{j''(j'' - n)p_2}''\).

3. Finally,

(a) if \(F = early F_1\), \(F = if F_1 \ then F_2 \ else F_3\), or \(F = F_1 \cdot F_2\), we have \(\alpha(F) = \alpha_S(F)\). Thus \(F\) is reducible to the normal form;

(b) otherwise, if \(F = F_1 \parallel F_2\), then \(\alpha(F_1) = \alpha(F_2) = true\). By induction assumption
we have

\[ F = F_1 \parallel F_2 \]

\[ = (\bigoplus_{k=1}^{n} \prod_{j=1}^{a_{k}} K_{jkl}) \parallel (\bigoplus_{k'=1}^{n'} \prod_{j'=1}^{a_{k'}} K'_{j'k'l'}) \]

\[ = \bigoplus_{k''=1}^{n''} \prod_{j''=1}^{a_{k''}} K''_{j''k''l''} \]

For any \( l'' \leq t \), we have \( K''_{j''k''l''} = K_{jkl} \), \( s'' = s \) and \( n'' = n \); otherwise, for any \( l'' > t \), we have \( K''_{j''k''l''} = K_{(l''-t)k''l''}, s'' = s' \) and \( n'' = n' \). □

### 4.4 Detecting communication interference

The abstract function \( \beta : (\mathcal{P} \cup \mathcal{I}) \rightarrow \mathbb{P}(\mathcal{X}) \) is presented to check interference for the first superstep of a given specification. In fact checking and analyzing the set of variables accessed during the first superstep are done at the same time. For example, if communication interference occurs between any two variables from the sets collected from two specifications, the parallel composition returns the infinite set \( \mathcal{X} \) to indicate the occurrence of interference. On the contrary, any interference-free specification produces a finite set of variables.

The interference relation is denoted: \( \bowtie \subseteq \mathcal{X} \times \mathcal{X} \). Primitive variables interfere iff they are identical. We use the operator \( \uplus \) to merge sets of variables.

\[
S_1 \uplus S_2 = \begin{cases} 
\mathcal{X} & \exists x_1 \in S_1, x_2 \in S_2 : (x_1 \bowtie x_2) \\
S_1 \cup S_2 & \text{otherwise}
\end{cases}
\]

For any \( \mathcal{X}_1, \mathcal{X}_2 \subseteq \mathcal{X} \), if there exist \( X_1 \in \mathcal{X}_1 \) and \( X_2 \in \mathcal{X}_2 \) such that \( X_1 \bowtie X_2 \) then \( \mathcal{X}_1 \uplus \mathcal{X}_2 = \mathcal{X} \); otherwise, \( \mathcal{X}_1 \uplus \mathcal{X}_2 = \mathcal{X}_1 \cup \mathcal{X}_2 \). It is obvious that the operator \( \uplus \) satisfies associativity and is distributive into \( \cup \).
Law 2  \[ \beta(P_1 \parallel P_2) = \beta(P_1) \uplus \beta(P_2) \quad \beta(S_1 ; S_2) = \beta(S_1) \]
\[ \beta(\text{if } E \text{ then } S_1 \text{ else } S_2) = \beta(S_1 \cap S_2) = \beta(S_1) \cup \beta(S_2) \]
\[ \beta(E \triangleright S) = \beta(S) \quad \beta(I_1 \otimes I_2) = \beta(I_1) \uplus \beta(I_2) \]
\[ \beta(\text{early } I) = \beta(I) \quad \beta(\text{after } x = E) = \{x\} \]

For example, \( \beta(P_{ex}) = \{x, y\} \). Note that since sequential processes in a nondeterministic choice or a binary conditional do not run at the same time, they will not interfere with each other.

**Proposition 2** The abstract function \( \beta \) satisfies all laws in Law 1.

**Proof.**

1. Need to prove the associativity of (\( \parallel \)), (\( ; \)), (\( \cap \)) and (\( \otimes \)). The proof is trivial, thus ignored.

2. Need to prove the distributivity of (\( ; \)) into (\( \cap \)).

\[ \beta((P_1 \cap P_2) \parallel P_3) \]
\[ = \]
\[ \beta(P_1 \cap P_2) \uplus \beta(P_3) \]
\[ = \]
\[ (\beta(P_1) \cup \beta(P_2)) \uplus \beta(P_3) \]
\[ = \]
\[ (\beta(P_1) \uplus \beta(P_3)) \cup (\beta(P_2) \uplus \beta(P_3)) \]
\[ = \]
\[ \beta(P_1 \parallel P_3) \cup \beta(P_2 \parallel P_3) \]
\[ = \]
\[ \beta((P_1 \parallel P_3) \cap (P_2 \parallel P_3)) \]

Similarly, we can prove \( \beta(P_3 \parallel (P_1 \cap P_2)) = \beta((P_3 \parallel P_1) \cap (P_3 \parallel P_2)). \)
3. Need to show that $\beta(\text{if } E \text{ then } S_1 \text{ else } S_2) = \beta((E \triangleright S_1) \cap (\neg E \triangleright S_2))$.

$$\begin{align*}
\beta(\text{if } E \text{ then } S_1 \text{ else } S_2) &= \\
&= \beta(S_1 \cap S_2) & \text{by } \beta \text{ definition} \\
&= \beta((E \triangleright S_1) \cap (\neg E \triangleright S_2))
\end{align*}$$

4. Need to show that $\beta(E \triangleright (S_1 \cap S_2)) = \beta((E \triangleright S_1) \cap (E \triangleright S_2))$.

$$\begin{align*}
\beta(E \triangleright (S_1 \cap S_2)) &= \\
&= \beta(S_1 \cap S_2) & \text{by } \beta \text{ definition} \\
&= \beta(S_1) \cup \beta(S_2) & \text{Obvious} \\
&= \beta((E \triangleright S_1) \cap (E \triangleright S_2)) & \text{by } \beta \text{ definition}
\end{align*}$$

5. Need to show that $\beta(E \triangleright (S_1 \triangleright S_2)) = \beta((E \triangleright S_1) \triangleright S_2)$.

$$\begin{align*}
\beta(E \triangleright (S_1 \triangleright S_2)) &= \\
&= \beta(S_1) & \text{by } \beta \text{ definition} \\
&= \beta(E \triangleright S_1) & \text{by } \beta \text{ definition}
\end{align*}$$
\[\beta((E > S_1) \triangleright S_2)\]

6. Need to show that \(\beta(E_1 \triangleright (E_2 > S)) = \beta((E_1 \land E_2) > S)\).

\[
\begin{align*}
\beta(E_1 \triangleright (E_2 > S)) &= \text{by}\ \beta\ \text{definition} \\
\beta(E_2 > S) &= \beta(S) \\
&= \text{by}\ \beta\ \text{definition} \\
\beta((E_1 \land E_2) > S)
\end{align*}
\]

7. Need to show that \(\beta(\text{true} > S) = \beta(S)\). The proof is trivial, thus ignored.

The function \(\beta\) satisfies all the laws. \(\square\)

**Lemma 1** A well-formed specification \(P\) contains communication interference in its first superstep iff \(\beta(P) = X\).

**Proof.** By Proposition 2 and Theorem 1, we are able to apply the normal form 4.3 on this function.

\[
\begin{align*}
\beta(P) &= \bigcup_{i \leq p} \beta(\|_{k \leq p} \Pi_{j \leq n} K_{jki}) \\
&= \bigcup_{i \leq p} \bigcup_{k \leq p} \beta(\Pi_{j \leq n} K_{jki}) \\
&= \bigcup_{i \leq p} \bigcup_{k \leq p} \beta(\Pi_{j \leq n} K_{1ki}) \\
&= \bigcup_{i \leq p} \bigcup_{k \leq p} \beta(\text{early} & \text{ after } X_{1kl} = E_{i1kl}, t_{1kl}) \\
&= \bigcup_{i \leq p} \bigcup_{k \leq p} \bigcup_{m \leq p} \{X_{1kl}\} \\
\end{align*}
\]

If the function \(\beta\) produces \(X\), by definition, there exists two variables: \(X_1\) and \(X_2\) such that \(X_1 \triangleq X_2\). Assume that these variables appears in the processes \(P_1\) and \(P_2\) respectively. We have

\[(\{X_1\} \subseteq \beta(P_1)) \land (\{X_2\} \subseteq \beta(P_2))\]
Since $\beta(P)$ presents the set of communication variables, we can conclude that those variables communicate with both $P_1$ and $P_2$, thus interference occurs. The proof is completed. □

Example 1

$\beta(P_{ex})$

$= \beta(\text{after } y = \text{before } x + 1) \land \beta(\text{after } x = \text{before } y + 2))$

$= \beta(\text{after } y) \land \beta(\text{after } x)$

$= \{y\} \land \{x\}$

$= \{y, x\}$

For a given specification, the abstract function $\gamma: \mathcal{P} \rightarrow \mathcal{P} \cup \{\text{II}\}$ removes its first superstep and extracts the tail. The tail of a specification with only one superstep is a special construct II called skip. If there are unbalanced processes, the translator can either report an error or simply ignore the shorter processes. We choose the latter approach in the following definition. For example, we have $\gamma(P_{ex}) = \text{II}$.

**Def 8**

$\gamma(P_1 \parallel P_2)$

$= \gamma(P_1) \parallel \gamma(P_2)$  \hspace{1cm} ($\gamma(P_1) \neq \text{II}, \gamma(P_2) \neq \text{II}$)

$\gamma(P_1 \parallel P_2)$

$= \gamma(P_1)$  \hspace{1cm} ($\gamma(P_2) = \text{II}$)

$\gamma(P_1 \parallel P_2)$

$= \gamma(P_2)$  \hspace{1cm} ($\gamma(P_1) = \text{II}$)

$\gamma(\text{if } E \text{ then } S_1 \text{ else } S_2)$

$= \gamma(S_1 \cap S_2)$

$\gamma(S_1 ; S_2)$

$= \gamma(S_1) ; S_2$  \hspace{1cm} ($\gamma(S_1) \neq \text{II}$)

$\gamma(S_1 ; S_2)$

$= S_2$  \hspace{1cm} ($\gamma(S_1) = \text{II}$)

$\gamma(S_1 \cap S_2)$

$= \gamma(S_1 \cap \gamma(S_2)$  \hspace{1cm} ($\gamma(S_1) \neq \text{II}, \gamma(S_2) \neq \text{II}$)

$\gamma(S_1 \cap S_2)$

$= \gamma(S_1)$  \hspace{1cm} ($\gamma(S_2) = \text{II}$)

$\gamma(S_1 \cap S_2)$

$= \gamma(S_2)$  \hspace{1cm} ($\gamma(S_1) = \text{II}$)

$\gamma(E \triangleright S)$

$= \gamma(S)$

$\gamma(\text{early } I)$

$= \text{II}$
Proposition 3 The abstract function $\gamma$ satisfies all laws in Law 1.

Proof. It can be proved like Proposition 2. □

Lemma 2 The abstract function $\gamma$ returns the tail of a given well-formed specification.

Proof. By Proposition 3 and Theorem 1, we are able to apply the normal form 4.3 on this function. Thus for any $P$, we have

$$
\gamma(P) = \prod_{l=1}^d \prod_{k=1}^e \prod_{n(s,t)}^{n(j)} K_{jkl}.
$$

The abstract function $\delta : \mathcal{P} \cup \{ \Pi \} \to \{ \text{true}, \text{false} \}$ combines $\beta$ and $\gamma$ and checks the whole specification superstep by superstep recursively: $\delta(S) = \text{true}$, $\delta(\Pi) = \text{true}$, $\delta(P) = \text{false}$ if $\beta(P) = \mathcal{X}$, and $\delta(P) = \delta(\gamma(P))$ otherwise. For example, $\delta(P_{ex}) = \text{true}$ indicating the freedom of communication interference in $P_{ex}$.

$$
\delta(P) = \begin{cases} 
\text{true} & P = \Pi \\
\text{false} & \beta(P) = \mathcal{X} \\
\delta(\gamma(P)) & \text{otherwise}
\end{cases}
$$

Theorem 4 A well-formed specification has communication interference iff the function $\delta$ returns false.

Proof. By definition, we have

$$
\delta(P) = (\beta(P) \neq \mathcal{X}) \land \delta(\gamma(P))
\quad = (\beta(P) \neq \mathcal{X}) \land (\beta(\gamma(P)) \neq \mathcal{X}) \land \delta(\gamma^2(P))
\quad = (\beta(P) \neq \mathcal{X}) \land \ldots \land (\beta(\gamma^n(P)) \neq \mathcal{X}) \land \delta(\Pi)
$$

By Proposition 2 and Lemma 6, the formulation $\beta(\gamma^i(P)) \neq \mathcal{X}$ checks communication interference for $i$-th super-step of the specification $P$. So the sequence of the formulations verify the inference from the first step to the last one. If one of them returns $\text{false}$, this whole specification has communication conflict. □
This theorem verifies specifications with multiple super-steps. The verification of the specifications becomes to verify a sequence of specifications with a single super-step. If there exist no interference in all the super-steps, the whole process has no communication conflict. Otherwise, the function δ returns false to indicate the interference.

4.5 Type checking

Simple (stateless) typing rules directly correspond to an abstract function. For example, Int and Bool are the types of integers and Boolean values. The following rules can be used to infer the types of expressions and directly correspond to implementation consisting of if-then-else conditional statements in structured programming style:

\[
\begin{align*}
E_1 &::= \text{Int} & E_2 &::= \text{Int} & E_1 + E_2 &::= \text{Int} \\
E_1 &::= \text{Bool} & E_2 &::= \text{Bool} & E_1 \land E_2 &::= \text{Bool} \\
\neg E &::= \text{Bool}.
\end{align*}
\]

Alternatively, we may regard types as (abstract) values and introduce functions for type calculation: \(\text{Int} \Rightarrow \text{Int} \land \text{Bool} \Rightarrow \text{Bool} \land \neg \text{Bool} \Rightarrow \text{Bool}\). Typing rules can then be modelled as an abstract function \(\epsilon : \mathcal{E} \rightarrow \text{TYPE}\): \(\epsilon(E_1 + E_2) = \epsilon(E_1) \lor \epsilon(E_2)\), \(\epsilon(E_1 \land E_2) = \epsilon(E_1) \land \epsilon(E_2)\) and \(\epsilon(\neg E) = \neg \epsilon(E)\). Unlike inference rules, abstract interpretation directly corresponds to implementation in Object-Oriented programming style with polymorphism. The abstract function can be implemented as an overloaded method without side effect.

Let \(\text{TYPE} \equiv \{\text{Int}, \text{Bool}, \text{True}, \text{False}\} \cup \text{RANGE}\) be the set of all types in LOGS. The inferred type of an expression is True or False if its truth-value can be determined statically; the inferred type is Bool if the precise type cannot be determined statically. \(\text{RANGE}\) is a set of interval types each \(\text{Range}(a, b)\) of which represents a range between integers \(a\) and \(b\) where \(a \leq b\). Interval types are used in range analysis (e.g. for array index). In particular, if the inferred type is \(\text{Range}(a, a)\), that means the dynamic value of the expression is constant and can be determined statically. The constant \(a\) will directly appear in the generated target code. Let \(\text{RANGE}_0\) represent
the set of these special range types. Clearly, it is the subset of RANGE. The additional definitions of type calculation are as follows:

\[
\text{Range}(a_1, b_1) + \text{Range}(a_2, b_2) = \text{Range}(a_1 + a_2, b_1 + b_2).
\]
\[
\text{Int} + \text{Range}(a, b) = \text{Range}(a, b) + \text{Int} = \text{Int}
\]
\[
\text{False} \lor T = T \lor \text{False} = \text{False} \lor \text{True} = \text{False}
\]
\[
\text{True} \lor T = T \lor \text{True} = T
\]
\[
\text{Bool} \land \text{Bool} = \text{Bool}
\]
\[
\neg \text{Bool} = \text{Bool}
\]

where \( T = \text{Bool}, \text{True}, \text{False} \). If an expression's type is undefined, in the FLEXIBO implementation, the translator program directly reports a runtime error, which is actually a compilation error for the source language LOGS.

The primitive types are primitive values in FLEXIBO. The rules of calculation are already embedded in the language and can be inherited by the translator. Many more types and arithmetic/logical/comparative operators are supported in FLEXIBO. RANGE is implemented as a class in the translator. Each object of the class has two attributes \( a \) and \( b \). Type calculation rules and type checking become the methods of the class.

The type of an expression may depend on a context, i.e. the state of the type checker. For example, the most accurate inferred type of an uninitialized variable is its declared type. After initializing it to an integer 1, its inferred type may be changed to \( \text{Range}(1, 1) \). This can be easily represented using state-dependent inference rules in the style of operational semantics. In abstract interpretation, it can be encoded as an additional argument of abstract functions. In FLEXIBO, it is directly implemented as a (polymorphic) method allowing side effects.

Let \( \rho : \mathcal{X} \rightarrow \text{TYPE} \) denote the mapping from variables to their types at the current point of static analysis, \( \mu : \mathcal{X} \rightarrow \mathcal{V} \) a mapping from variables to their initial values, \( \tau : \mathcal{X} \rightarrow \text{TYPE} \) a mapping from variables to their declared types, and \( \pi : \mathcal{V} \rightarrow \text{TYPE} \) a mapping from constant values to their inferred types. We assume \( \pi(n) = \text{Range}(n, n) \) for any integer \( n \), \( \pi(\text{true}) = \text{True} \) and \( \pi(\text{false}) = \text{False} \).
The abstract function $\epsilon : \mathcal{E} \rightarrow \text{TYPE}$ evaluates the type of an expression in a given state $\rho$.

**Def 9**

$\epsilon(E_1 + E_2) = \epsilon(E_1) \oplus \epsilon(E_2)$

$\epsilon(E_1 \land E_2) = \epsilon(E_1) \land \epsilon(E_2)$

$\epsilon(\neg E) = \neg \epsilon(E)$

$\epsilon(\text{before } x) = \tau(x)$

$\epsilon(x) = \rho(x)$

$\epsilon(v) = \pi(v)$

Note that the inferred type of a variable of communication (in $\text{before } y$) is always its declared type $\tau(x)$, while that of an independent variable is context-related. For example, suppose $\tau(x) = \text{Int}$, then $\epsilon(\text{before } y + 1) = \tau(y) \land \text{Range}(1, 1) = \text{Int}$. Another example is

$\epsilon(1 + 1) = \text{Range}(1, 1) \land \text{Range}(1, 1) = \text{Range}(2, 2)$

Before defining the type checking for LOGS, we introduce a subtyping partial ordering $\ll$. For example, the command $\text{after } x = E$ attempts to write to variable $x$ remotely. It requires that the inferred type of $E$ be a subtype of the declared type of $x$. We assume that $\text{Range}(a_1, b_1) \ll \text{Range}(a_2, b_2) \ll \text{Int}$ if $a_2 \leq a_1 \leq b_1 \leq b_2$ and $\text{True}, \text{False} \ll \text{Bool}$. In addition, the abstract function $\epsilon : \mathcal{P} \cup \mathcal{I} \rightarrow \{ \text{true}, \text{false} \}$ checks whether a specification is type-consistent in a given state of static analysis.

**Def 10**

$\epsilon(P_1 \parallel P_2) = \epsilon(P_1) \land \epsilon(P_2)$

$\epsilon(S_1 \mathbin{;} S_2) = \epsilon(S_1) \land \epsilon(S_2)$

$\epsilon(\text{if } E \text{ then } S_1 \text{ else } S_2) = \epsilon(E) \ll \text{Bool} \land \epsilon(S_1) \land \epsilon(S_2)$

$\epsilon(S_1 \cap S_2) = \epsilon(S_1) \land \epsilon(S_2)$

$\epsilon(E \triangleright S) = \epsilon(E) \ll \text{Bool} \land \epsilon(S)$

$\epsilon(\text{early } I) = \epsilon(I)$

$\epsilon(T_1 \odot T_2) = \epsilon(T_1) \land \epsilon(T_2)$

$\epsilon(\text{after } x = E) = \epsilon(E) \ll \tau(x)$

The condition in a conditional must be a boolean, and the inferred type of the expression $E$ must be a subtype of the declared type of the accessed variable $x$ in $\text{after } x = E$.

**Proposition 5** The abstract function $\epsilon$ satisfies all laws in Law 1.
Proof. Here only present the proof for the law(5) and the law(6). Others can be easily completed.

1. Need to show that $\varepsilon(E \triangleright (S_1 \triangleright S_2)) = \varepsilon((E \triangleright S_1) \triangleright S_2)$.

$$\varepsilon(E \triangleright (S_1 \triangleright S_2))$$
$$= \varepsilon(E) \triangleleft \text{Bool} \land \varepsilon(S_1 \triangleright S_2)$$
$$= \varepsilon(E) \triangleleft \text{Bool} \land \varepsilon(S_1) \land \varepsilon(S_2)$$
$$= \varepsilon((E \triangleright S_1) \triangleright S_2)$$

2. Need to show that $\varepsilon(E_1 \triangleright (E_2 \triangleright S)) = \varepsilon((E_1 \land E_2) \triangleright S)$.

$$\varepsilon(E_1 \triangleright (E_2 \triangleright S))$$
$$= \varepsilon(E_1) \triangleleft \text{Bool} \land \varepsilon(E_2 \triangleright S)$$
$$= \varepsilon(E_1) \triangleleft \text{Bool} \land \varepsilon(E_2) \triangleleft \text{Bool} \land \varepsilon(S)$$
$$= \varepsilon((E_1 \land E_2) \triangleright S)$$

The proof is completed. □
In FLEXIBO, LOGS type checking is implemented as a (polymorphic) method that returns a Boolean value. We assume $\rho = \tau$ for basic specifications, as they do not modify the types of variables. For example, if $x$ and $y$ are declared as integers, we then have:

$$
\varepsilon(P_{ex}) = \varepsilon(\text{before } y + 1) \preceq \tau(x) \land \varepsilon(\text{before } x + 2) \preceq \tau(y) \\
= (\text{Int } \uparrow \text{Range}(1, 1)) \preceq \text{Int} \land (\text{Int } \uparrow \text{Range}(2, 2)) \preceq \text{Int} \\
= \text{Int } \preceq \text{Int} \land \text{Int } \preceq \text{Int} \\
= \text{true}.
$$

### 4.6 Variable registration for communication

BSPlib programs needs to register the variables involved in communication (either read or written) at the beginning of each process.

We first define an abstract function $\omega : E \rightarrow \mathcal{P}(\mathcal{X})$ that collects the set of all variables to be read in a given expression.

**Def 11**

$\omega(E_1 + E_2) = \omega(E_1) \cup \omega(E_2)$

$\omega(E_1 \land E_2) = \omega(E_1) \cup \omega(E_2)$

$\omega(\neg E) = \omega(E)$

$\omega(\text{before } x) = \{ x \}$

$\omega(y) = \{ \}$

To generate the code for variable registration, the abstract function $\zeta : (\mathcal{P} \cup I) \rightarrow \mathcal{P}(\mathcal{X})$ to collect the set of all variables that are either in after $x$ or before $x$. For example, $\zeta(P_{ex}) = \{ x, y \}$.

**Def 12**

$\zeta(P_1 \parallel P_2) = \zeta(P_1) \cup \zeta(P_2)$

$\zeta(S_1 \uplus S_2) = \zeta(S_1) \cup \zeta(S_2)$

$\zeta(\text{if } E \text{ then } S_1 \text{ else } S_2) = \zeta(S_1) \cup \zeta(S_2)$

$\zeta(S_1 \cap S_2) = \zeta(S_1) \cup \zeta(S_2)$

$\zeta(E \triangleright S) = \omega(E) \cup \zeta(S)$

$\zeta(\text{early } I) = \zeta(I)$

$\zeta(I_1 \otimes I_2) = \zeta(I_1) \cup \zeta(I_2)$

$\zeta(\text{after } x = E) = \{ x \} \cup \omega(E)$
Proposition 6  The abstract function $\zeta$ satisfies all the laws in Law 1.

Proof. Only need to prove the law(6) in Law 1. $\zeta(E_1 \triangleright (E_2 \triangleright S))$

$= \omega(E_1) \cup \omega(E_2) \cup \zeta(S)$

$= \omega(E_1 \land E_2) \cup \zeta(S)$

$= \zeta((E_1 \land E_2) \triangleright S)$

The proof is completed. □

Theorem 7  The set of all variables in after $x$ and before $x$ of a well-formed specification $P$ is $\zeta(P)$.

Proof. By Proposition 6 and Theorem 1, we are able to apply the normal form 4.3 on this function. Thus, for any $P$, we have

$$\zeta(P) = \bigcup_{ijkl} (\omega(E_{ijkl}) \cup \{x_{ijkl}\} \cup \omega(E_{ijkl}))$$

□

4.7  Analysis of communication dependencies

If a process writes a new value to a variable, it features a bsp.put communication to every process that reads it. In this chapter, we assume that the writing process will send the new value to every process that may ever read the variable (not just in the subsequent supersteps).

Before analyzing the dependencies, we use an abstract function $\theta : P \rightarrow N$ to account the total number of sequential processes in a specification. For example, $\theta(P_{cz}) = 2$. 
Def 13 \[ \theta(P_1 \parallel P_2) = \theta(P_1) + \theta(P_2) \]
\[ \theta(S) = 1 \]

Proposition 8 The abstract function \( \theta \) satisfies all the laws in Law 1.

Proof. Obvious \[ \square \]

Theorem 9 The total number of sequential processes in any well-formed specification \( P \) is \( \theta(P) \).

Proof. By Proposition 6 and Theorem 1, we are able to apply the normal form 4.3 on this function. Thus, for any \( P \), we have \( \theta(P) = t \). \[ \square \]

The process \( id \) of the communication destination must be identified for a \texttt{bsp.put} command.

We use an abstract function \( \eta : \mathbb{N} \times \mathbb{N} \times (\mathcal{P} \cup \mathcal{I}) \to \mathcal{P}(\mathcal{X}) \) to collect the set of all variables (in \texttt{before} \( x \) of process \( i \)) to be read. The first argument is the absolute process \( id \), the second argument is the relative starting process \( id \) of the specification, and the last argument is a specification that may include several sequential processes. This design guarantees the compositionality for the operation.

Def 14 \[ \eta(i, n, P_1 \parallel P_2) = \eta(i, n, P_1) \cup \eta(i, n + \theta(P_1), P_2) \]
\[ \eta(i, n, S) = \{ \} \quad (i \neq n) \]
\[ \eta(i, n, \text{if } E \text{ then } S_1 \text{ else } S_2) = \omega(E) \cup \eta(i, n, S_1 \cap S_2) \quad (i = n) \]
\[ \eta(i, n, S_1 \cap S_2) = \eta(i, n, S_1 ; S_2) = \eta(i, n, S_1) \cup \eta(i, n, S_2) \quad (i = n) \]
\[ \eta(i, n, E \triangleright S) = \omega(E) \cup \eta(i, n, S) \quad (i = n) \]
\[ \eta(i, n, \text{early } I) = \eta(i, n, I) \quad (i = n) \]
\[ \eta(i, n, I_1 \otimes I_2) = \eta(i, n, I_1) \cup \eta(i, n, I_2) \quad (i = n) \]
\[ \eta(i, n, \text{after } x = E) = \omega(E) \quad (i = n) \]

For example, \( \eta(i, i, P_{ex}) = \{ y \} \), \( \eta(i, i - 1, P_{ex}) = \{ x \} \), but \( \eta(i, n, P_{ex}) = \{ \} \) if \( n \not\in \{ i, i + 1 \} \).
Proposition 10 The function \( \eta \) satisfies all laws in Law 1.

Proof.

1. Need to prove the associativity of \((\cdot \| \cdot)\), \((\cdot \circ \cdot)\), \((\cdot \cdot \cdot)\) and \((\cdot \otimes \cdot)\). We only state the associativity of \((\cdot \| \cdot)\).

\[
\eta(i, n, P_1 \| (P_2 \| P_3)) = \eta(i, n, P_1) \cup \eta(i, n + \theta(P_1), (P_2 \| P_3)) \\
= \eta(i, n, P_1) \cup \eta(i, n + \theta(P_1), P_2) \cup \eta(i, n + \theta(P_1) + \theta(P_2), P_3) \\
= \eta(i, n, P_1 \| P_2) \cup \eta(i, n + \theta(P_1) + \theta(P_2), P_3) \\
= \eta(i, n, (P_1 \| P_2) \| P_3).
\]

2. Need to prove the distributivity of \((\cdot \circ \cdot)\) into \((\cdot \cdot \cdot)\). The proof is trivial, thus ignored.

3. Need to show that \(\eta(i, n, \text{if } E \text{ then } S_1 \text{ else } S_2) = \eta(i, n, E \supset S_1) \cap (\neg E \supset S_2))\).

\[
\eta(i, n, \text{if } E \text{ then } S_1 \text{ else } S_2)) \\
= \omega(E) \cup \eta(i, n, (S_1 \cap S_2)) \\
= \omega(E) \cup \eta(i, n, S_1) \cup \omega(E) \cup \eta(i, n, S_2) \\
= \eta(i, n, E \supset S_1) \cup \eta(i, n, \neg E \supset S_2) \\
= \eta(i, n, (E \supset S_1) \cap (\neg E \supset S_2))
\]

4. Need to show that \(\eta(i, n, E \supset (S_1 \cap S_2)) = \eta(i, n, (E \supset S_1) \cap (E \supset S_2))\).

\[
\eta(i, n, E \supset (S_1 \cap S_2)) \\
= \omega(E) \cup \eta(i, n, S_1 \cap S_2)
\]
by η definition

\[ \omega(E) \cup \eta(i, n, S_1) \cup \omega(E) \cup \eta(i, n, S_2) \]

by η definition

\[ \eta(i, n, E \triangleright S_1) \cup \eta(i, n, E \triangleright S_2) \]

by η definition

\[ \eta(i, n, (E \triangleright S_1) \cap (E \triangleright S_2)) \]

by η definition

5. Need to show that \( \eta(i, n, E \triangleright (S_1 \uplus S_2)) = \eta(i, n, (E \triangleright S_1) \uplus S_2) \).

\[ \eta(i, n, E \triangleright (S_1 \uplus S_2)) \]

by η definition

\[ \omega(E) \cup \eta(i, n, S_1) \cup \eta(i, n, S_2) \]

by η definition

\[ \eta(i, n, (E \triangleright S_1) \uplus S_2) \]

by η definition

6. Need to show that \( \eta(i, n, E_1 \triangleright (E_2 \triangleright S)) = \eta(i, n, (E_1 \triangleright E_2) \triangleright S) \).

\[ \eta(i, n, E_1 \triangleright (E_2 \triangleright S)) \]

by η definition

\[ \omega(E_1) \cup \omega(E_2) \cup \eta(i, n, S) \]

by η definition

\[ \omega(E_1 \land E_2) \cup \eta(i, n, S) \]

by η definition

\[ \eta((E_1 \land E_2) \triangleright S) \]

by η definition

7. Need to show that \( \eta(i, n, \text{true} \triangleright S) = \eta(i, n, S) \). The proof is trivial, thus ignored.

\[ \square \]
Theorem 11 The abstract function \( \eta(i, n, P) \) collects the set of all variables read by process \( i \), a sequential process in a well-formed specification \( P \) with starting process id \( n \).

Proof. By Proposition 27 and Theorem 1, we are able to apply the normal form 4.3 on this function. Thus, for any \( P \), if \( n' \leq i' < n' + t \), we let \( l \equiv i' - n' + 1 \), and then \( \eta(i', n', P) = \bigcup_{ijk} \omega(E_{ijkl}) \); otherwise, \( \eta(i', n', P) = \{ \} \). □

A process writing to a shared variable must issue a `basp.put` communication to every process that may read another variable interfering with the shared variable.

4.8 Generating code for expressions

The abstract function \( \phi : E \rightarrow \text{String} \) is presented to translate a LOGS expression into a C expression (as a string). The operation \( \text{str}_1 + \text{str}_2 \) denotes string concatenation.

If the dynamic value of an sub-expression can be determined statically in type calculation, the generated code for that sub-expression will be the static value itself; otherwise, the sub-expression is converted to a string in C syntax. If the inferred type is a singleton range: \( \epsilon(E) = \text{Range}(a, a) \), then we let \( \phi(E) \) be "a", a string converted from the integer \( a \); if \( \epsilon(E) = \text{True} \) or \( \epsilon(E) = \text{False} \), then \( \phi(E) = "1" \) (i.e. the boolean \text{true} in C) or \( \phi(E) = "0" \) (i.e. the boolean \text{false} in C) respectively; otherwise, the abstract function follows the following definition:

\[
\text{Def 15} \quad \phi(E_1 + E_2) = \phi(E_1) + " + " + \phi(E_2) \\
\phi(E_1 \land E_2) = \phi(E_1) + " \land " + \phi(E_2) \\
\phi(\neg E) = " ! " + \phi(E) \\
\phi(\text{before } x) = \phi(x) = "x" \\
\phi(v) = "v".
\]
For example, if \( \tau(y) = \text{Int} \), then \( \phi(\text{before } y + 1) = "y + 1" \), as \( \epsilon(\text{before } y + 1) = \text{Int} \rightarrow \text{Range}(1, 1) = \text{Int} \); on the other hand, \( \phi(1 + 1) = "2" \), because its value can be determined statically in type calculation: \( \epsilon(1 + 1) = \text{Range}(1, 1) \rightarrow \text{Range}(1, 1) = \text{Range}(2, 2) \).

### 4.9 Generating code for communications

The code for a sequential process that writes into a shared variable may involve other processes in parallel. Thus the abstract function \( \psi : (S_0 \cup \mathcal{I}) \times \mathcal{P} \rightarrow \text{String} \) of communication code generation must have the whole specification as an argument. Let \( \lambda(x, P) \) denote the set of id numbers of the processes that may read variable \( x \):

\[
\lambda(x, P) = \{i < \theta(P) \mid \exists y \in \eta(i, 0, P) \cdot x \Rightarrow y\}.
\]

For example, we have \( \lambda(x, P_{ex}) = \{1\} \), \( \lambda(y, P_{ex}) = \{0\} \), and for any other variable \( z \), \( \lambda(z, P_{ex}) = \{\} \). In the following definition, we use \( \int \) to denote collective string concatenation. For example, \( \int_{i \in \{1, 3, 4\}} str_i = str_1 + str_3 + str_4 \). Note that code generation does not have to deal with non-deterministic choice and conditional magic, which are by-products of static analysis.

**Def 16**

\[
\psi(\text{if } E \text{ then } S_1 \text{ else } S_2, P) = "\text{if" } + \phi(E) + "\{" + \psi(S_1, P) + "\}" + "\text{else" } + \psi(S_2, P) + "\}"
\]

\[
\psi(S_1 ; S_2, P) = \psi(S_1, P) + \psi(S_2, P)
\]

\[
\psi(\text{early } I, P) = \psi(I, P) + "\text{bsp___sync()};"
\]

\[
\psi(I_1 \otimes I_2, P) = \psi(I_1, P) + \psi(I_2, P)
\]

\[
\psi(\text{after } x = E, P) = "x := " + \phi(E) + "\};" + \int_{i \in \lambda(x, P)} "\text{bsp___put}(i, x, x, 0, \text{sizeof}(\tau(x)));"
\]

For example, \( \psi(\text{early} (\text{after } x = \text{before } y + 1), P_{ex}) \) generates C code:

\[
x := y + 1; \text{ bsp\_put(1, x, x, 0, sizeof(int)); bsp\_sync();}
\]
The actual implementation of the above abstract function is a (polymorphic) method that directly prints out the code as output.

4.10 Generating code for processes

A sequential process starts from the `bsp.pushregister` of shared variables and ends with their release by `bsp.popregister`. The abstract function \( \kappa : (N \times P \times P) \to \text{String} \) generates the code for each process \( n \). The first argument is the relative starting process id, the second one is the sub-specification to be translated, and the last one is the whole specification.

\textbf{Def 17}

\[
\kappa(n, P_1 \parallel P_2, P) = \kappa(n, P_1, P) + \kappa(n + \rho(P_1), P_2, P)
\]

\[
\kappa(n, S, P) = \text{"void process } n \{ " + \int_{x: \zeta(P)} \text{"bsp.pushregister(\&x, size of } \tau(x)\text{);} + \psi(S, P) + \int_{x: \zeta(P)} \text{"bsp.popregister(\&x);} + \text{"} \}
\]

If a specification \( P \) passes syntactical checking, communication interference checking and type checking, the target C code can be generated in the following structure:

\[
\text{"#include ..."} + \kappa(0, P, P) + \text{"void main() ..."}
\]

where we assume that \( \rho = \tau \).

4.11 Summary

This chapter presents the basic static-analysis methods for generating C code from LOGS specifications. The syntax adopted is restrictive, but supports powerful static-analysis methods and generates
fast and safe BSPlib-C code. The technique of normal form proves to be an illustrative tool for verifying the correctness of the translator. This chapter is to demonstrate the construction of translators and their formal verification. The translator provides useful assistance to human programmers by automatically adding communication commands, generating code for each process, optimizing expressions and partitioning data structures. Although this chapter is primarily presented in abstract interpretation, the actual implementation sometimes "conveniently" deviates from the style of denotational semantics. For example, it is convenient to allow side effect on the state of the static analyzer during type checking, which is conveniently presented using inference rules in the style of operational semantics. Nevertheless both styles can be straightforwardly supported by Object-Oriented programming. This perhaps highlights the integrating power of OO framework for static analysis. The translator generates a separate code for every processor.
Chapter 5

Generating BSPlib-C code from LOGS Specifications(II)

5.1 Introduction

The previous chapter [96] investigates the transformation for the basic LOGS and formally states its correctness. However, that primitive specification language lacks the support to present more expressive programs such as SIMD, iteration, etc. To improve programming expressivity, this chapter extends that basic LOGS with a number of advanced instructions.

SIMD parallelism corresponds to a command \( \text{par} \ x \ \text{from} \ E_1 \ \text{to} \ E_2 \ \text{do} \ S. \) The values of the expressions \( E_1 \) and \( E_2 \) must be natural numbers statically determinable during type calculation; otherwise, a typing error is reported. The command generates a number of sequential processes in parallel in the target code. For each process, the variable \( x \) takes a different singleton range \( \text{Range}(a, a) \) where \( a \) is any number between the values of \( E_1 \) and \( E_2 \). Communication interference, type consistency and communication dependencies are checked for each process independently.

In order to support static analysis, the translator only supports a restricted form of iteration \( \text{for} \ x \ \text{from} \ E_1 \ \text{to} \ E_2 \ \text{do} \ S. \) Again, \( E_1 \) and \( E_2 \) must be statically determinable. Such a command always terminates and can be translated to a for-loop in C. Communication interference is checked
superstep by superstep after unfolding the iteration. If there are too many supersteps, a decision must be made on a limit $n$: the static analyser will only check the freedom of communication interference for the first $n$ supersteps. Types and communication dependencies, on the other hand, are analysed without unfolding the iteration. The range between the values of $E_1$ and $E_2$ will be used as the inferred type of $x$ recorded in the new state of the static analyser. The body of the for-loop will be type-checked under the abstraction.

The operator $x\#E$ represents the access of array $x$ at the index $E$. Range analysis is important to guarantee valid access for index expression. Two array accesses interfere if they are on the same array and the type-inferred ranges of the index expressions overlap each other. Note that the interface detection is based on abstraction rather than accurate computation. For simplification, we only discuss single-dimension arrays here.

With the advanced instructions, the extended LOGS possesses more complicated syntax. The primitive LOGS can be generally considered as a sub-langauge with a simplified normal form. It is provable that the new normal form involving environments is able to represent all the well-formed programs constructed with those basic instructions. The extended LOGS may change the static analyzer's state. Thus advanced specifications have an extended normal form incorporating additional commands that allows the state of the analyzer to be changed before each early transition. Both commands can then be decomposed into the normal form with that additional command.

Similar to the approach shown in the previous chapter, a set of assumptions are proposed to guarantee that well-formed programs collapse to the extended normal form. Apart from the normal ones, additional assumptions are mainly related to environments. In addition, a couple of laws ensures that the instructions: par and for are reducible to the forms constructed with basic commands.

In response to the additional instructions and extended normal form, most abstract functions need to be redefined to involve extra arguments for environments. For instance, the function $\beta$ for communication interference detection corresponds to $\beta^*$, which is a context-dependent abstract function for interference checking. Similarly, other functions (e.g. $\gamma^*$) with the mark "*" redefine
the corresponding context-free ones (e.g. \( \gamma \)). Of course, we reuse the functions, which are completely context-independent.

The translation for the extended language also needs to take a number of steps. The transformer firstly checks syntactical well-formness for a given specification, then detects (actually approximates) potential communication interference, analyzes type consistency, finally generates the corresponding target code.

After syntactical checking, we assume that further abstract functions only deal with the well-formed programs. For instance, the validation in syntax guarantees that the index variables of `par` and `for` are primitive and the bounds are reduced to natural numbers. The further functions like communication interference detection operate on well-formed programs.

The renewed abstract function of communication interference detection builds on \( \beta \), but involves the extra argument to represent the context. It is provable that for the programs in that basic LOGS, the renewed function \( \beta^* \) produces the same result as the original one \( \beta \). Therefore the function is able to correctly calculate the programs in that basic LOGS. Note that \( \beta^* \) is not able to accurately determine the interference, but approximately because we may not statically determine the explicit index for array access.

As presented previously, the language only deals with finite iteration. Therefore, for the instruction `par`, the abstract function to get its tail returns a new iteration with an increased lower bound. The instruction `par` can be generally considered as a set of parallel processes with each individual environment. Thus its tail is a new `par` constructed by the tails of each its parallel processes.

Code generation for a specification needs to register and release shared variables, features the commands `(bsp.put)` for variable communication, and finally produce a corresponding target program. Although element selections are generally regarded as variables, they will not be registered individually, but wholly as an array. For `par` and `for`, variable registration only happens to their bodies. The output of an expression may be either a target expression (as a string) or a concrete
value. The style depends on the type of the expression. If its dynamic value can be determined statically, the value is converted to a string as the output. Otherwise, it is directly printed as a expression in the target language.

5.2 Syntactical checking

To present more expressive programs, the extended language LOGS corresponds to the following syntax in FLEXIBO.

\[
F ::= \text{par } F \text{ from } F \text{ to } F \text{ do } F \mid F \mid F ; F \mid \text{for } F \text{ from } F \text{ to } F \text{ do } F \\
\mid \text{if } F \text{ then } F \text{ else } F \mid \text{early } F \mid F \otimes F \mid \text{after } F \leftarrow F \mid F + F \\
\mid F \wedge F \mid F \# F \mid \neg F \mid \text{before } F \mid x \mid v.
\]

The instruction \text{par } F \text{ from } F \text{ to } F \text{ do } F presents SIMD-style parallelism. The command \text{for } F \text{ from } F \text{ to } F \text{ do } F stands for a restricted form of finite iteration. The operator \text{F} \# \text{F} is used to select elements from an array.

LOGS possesses more restrictions in syntax. The above FLEXIBO's syntax has the following hierarchy in LOGS.

\[
P ::= \text{par } N \text{ from } E \text{ to } E \text{ do } S \mid P \parallel P \mid S \\
S ::= S ; S \mid \text{for } N \text{ from } E \text{ to } E \text{ do } S \mid \text{if } E \text{ then } S \text{ else } S \\
\mid \rho \leftarrow S \mid S \cap S \mid E \triangleright S \mid T \\
T ::= \text{early } I \\
I ::= I \otimes I \mid W \\
W ::= A = E \\
A ::= \text{after } X \\
E ::= E + E \mid E \wedge E \mid \neg E \mid \text{before } X \mid X \mid V \\
X ::= x \mid X \# E \\
N ::= x \\
V ::= v.
\]

Note that the extended LOGS syntax generally regards element selections as variables, but the index variables in the instructions: \text{par} and \text{for} must be primitive variables. Although this syntax allows
more complicated structures for \( X \# E \), the chapter only studies the non-recursive form for simplification. In the other word, we only discuss the selection for one-dimension array. The by-product operator \( \rightarrow \) does not directly involve in LOGS program, but in static analysis and normal form. The term \( \rho \) is the set to record a number of mappings from variables to their real types. The following program presents a well-formed specification in LOGS.

\[
P_{ex} \triangleq (\text{for } w \text{ from } 1 \text{ to } 2 \text{ do (after } x = \text{ before } y \# w + 1)) \quad \| \quad (\text{par } v \text{ from } 1 \text{ to } 3 \text{ do early (after } y = \text{ before } x + 2))
\]

In response to the additional instructions, we define the following boolean functions to formalize their syntactical checking.

**Def 18**

\[
\begin{align*}
\alpha_{(\text{par } F_1 \text{ from } F_2 \text{ to } F_3 \text{ do } F_4)} & = \alpha_N(F_1) \land \alpha_E(F_2) \land \alpha_E(F_3) \land \alpha_S(F_4) \\
& \land (\tau(F_1) = \text{Int}) \land (\{\epsilon(F_2), \epsilon(F_3)\} \subseteq \text{RANGE}_0) \\
\alpha_{(\text{for } F_1 \text{ from } F_2 \text{ to } F_3 \text{ do } F_4)} & = \alpha_N(F_1) \land \alpha_E(F_2) \land \alpha_E(F_3) \land \alpha_S(F_4) \\
& \land (\tau(F_1) = \text{Int}) \land (\{\epsilon(F_2), \epsilon(F_3)\} \subseteq \text{RANGE}_0) \\
\alpha_E(F_1 \# F_2) & = \alpha_X(F_1 \# F_2) \\
\alpha_X(F_1 \# F_2) & = \alpha_X(F_1) \land \alpha_E(F_2) \\
\alpha_N(x) & = true
\end{align*}
\]

The instructions: \text{par} and \text{for} requires that the first expressions must be primitive variables, both \( F_2 \) and \( F_3 \) are reducible to integer. The abstract function \( \alpha_N \) checks whether an expression is a primitive variable. Further discussion only deals with well-formed specifications. For convenience, let \( P_r \) and \( F_r \) denote \text{par } \( x \text{ from } E \text{ to } E \text{ do } S \) and \text{for } \( x \text{ from } E \text{ to } E \text{ do } S \) respectively. The functions above are able to validate the syntax for a given program. For instance, \( \alpha(P_{ex}) \) produces \text{true}. It indicates the the sample program succeeds in passing syntactical checking.

**FLEXIBO's** program for syntactical checking can be simplified as follows. These methods directly correspond to the above formalization.
var LOGS := class Reflection { 
  var SemExp := class (superclass.SemExp) { 
    var alphaP := method [] false;
    var alphaS := method [] false;
    var alphaE := method [] false;
    var alphaX := method [] false;
    var alphaN := method [] false;
  };
  var SemPar := class (superclass.SemForAll) { 
    var alphaP := method []
        variable.alphaN[] && e1.alphaE[] && e2.alphaE[] && body.alphas[];
  };
  var SemForAll := class (superclass.SemForAll) { 
    var alphaP := method [] alphaS[];
    var alphaS := method [] 
        variable.alphaN[] && e1.alphaE[] && e2.alphaE[] && body.alphas[];
  };
  var SemMethodInvoke := class (superclass.SemForAll) { 
    var alphaE := method [] alphaX[];
    var alphaX := method [] e1.alphaX[] && e2.alphaE[];
  };
  var PubStaticVariable := class (superclass.PubStaticVariable) { 
    var alphaE := method [] alphaX[];
    var alphaN := method [] alphaX[];
    var alphaX := method [] true;
  };
}; LOGS.flexibo[];

The program firstly defines all the methods to return false by default in SemExp. As the subclasses
of SemExp, other classes just need to redefine some of those methods, and inherit the rest. In the
classes: SemPar and SemForAll, the object variable stands for the index expression, similarly
body for the body.

5.3 Advanced assumptions, normal form and completeness

To present the advanced instructions, we need to renew the previous normal form. The following
additional laws are assumed to be satisfied for the extended LOGS programs.

**Law 3 (Advanced assumptions)**

1. \( \rho_1 \rightarrow (\rho_2 \rightarrow S) = (\rho_1 \uparrow \rho_2) \rightarrow S \)
2. \( E \Rightarrow (\rho \rightarrow S) = \rho \rightarrow (E \Rightarrow S) \)
3. \( \rho \rightarrow (S_1 \uplus S_2) = (\rho \rightarrow S_1) \uplus S_2 \)
4. \( \emptyset \rightarrow S = S \)
5. \( \text{par } x \text{ from } E_1 \text{ to } E_2 \text{ do } S \) can be reduced to a plain form.
6. \( \text{for } x \text{ from } E_1 \text{ to } E_2 \text{ do } S \) can be reduced to a plain form.

A plain form is the one that is represented by the basic commands (excluding par and for). The
last two assumptions do not require concretely mathematical definitions. They loosely require that
these instructions are reducible to plain forms. Further abstract functions have different definitions
for them. The weak requirements are fairly enough to guarantee to produce a normal form. The
term \( \emptyset \) represents an empty mapping set. For any environment \( \rho \), we always have \( \rho \uparrow \emptyset = \emptyset \uparrow \rho = \rho \).

With the assumptions in **Law 1** and **Law 3**, LOGS programs collapses to the following normal
form.
Unlike the previous one, the form involves an environment and allows more general variables (e.g., element selection) to follow the keyword `after`. It can be merged into a single line. Parallel compositions choices are located in the outmost layer, and then nondeterministic, sequential compositions, environments, conditionals and finally early transitions.

$$t = \prod_{i=1}^{m} \text{after } X_i = E_i$$

where $t$ is a constant, but $s = s(t)$, $n = n(s, t)$ and $m = m(n, s, t)$ are dependent functions.

**Theorem 12 (Completeness of assumptions)** Any well-formed LOGS specification can be reduced to the above normal form under the assumptions of Law 1 and Law 3.

**Proof.** We prove by induction on the structure of program constructs.

1. Need to show that, if a program construct $F$ satisfies $\alpha_1(F) = \text{true}$, it can be reduced to the normal form $\otimes_{i=1}^{m} \text{after } X_i = E_i$:

   (a) indeed, if $F = (\text{after } F_1 = F_2)$, and hence $\alpha_X(F_1) = \text{true}$ and $\alpha_E(F_2) = \text{true}$, then $F = \otimes_{i=1}^{m} \text{after } X_i = E_i$ where $X_1 = F_1$ and $E_1 = F_2$;

   (b) otherwise, if $F = F_1 \otimes F_2$ where $\alpha_1(F_1) = \alpha_1(F_2) = \text{true}$, then by induction assumption, we have $F_1 = \otimes_{i=1}^{m} \text{after } X_i = E_i$ and $F_2 = \otimes_{i=1}^{m'} \text{after } X_{i'} = E'_{i'}$, and thus $F = \otimes_{i'=1}^{m+m'} \text{after } X_{i''} = E_{i''}$ where for any $i'' \leq m$, $X_{i''} = X_i$ and $E_{i''} = E_i$; and for any $i'' > m$, $X_{i''} = X_{i''-m}$ and $E_{i''} = E_{i''-m}$. 


2. Need to present that, if a program construct $F$ satisfies $\alpha_S(F) = true$, it can be reduced to the normal form $F = \prod_{k=1}^{n} \prod_{j=1}^{m} \rho_{jk} \rightarrow K_{jk}$:

(a) if $F = (early F_1)$, and hence $\alpha_j(F_1) = true$. We have $F_1 = \bigotimes_{i=1}^{m} after X_i = E_i$, hence $F = T$ and $T = (true \triangleright T)$. Therefore, by the assumption(4) in Law 3, we have

$$F = \prod_{k=1}^{1} \prod_{j=1}^{1} \emptyset \rightarrow (true \triangleright T_{jk}) = \prod_{k=1}^{1} \prod_{j=1}^{1} \emptyset \rightarrow K_{jk}$$

(b) if $F = (if F_1 then F_2 else F_3)$, thus according to the assumption(3) in Law 1, we have $F = (E \triangleright S_1) \cap (\neg E \triangleright S_2)$.

$\alpha_E(E) = \alpha_S(S_1) = \alpha_S(S_2) = true$, by induction assumption, let $E$ be an expression $E \in E$ and $S_1, S_2$ be the expressions as follows:

$S_1 = \prod_{k=1}^{a} \prod_{j=1}^{a} \rho_{jk} \rightarrow K_{jk} \quad S_2 = \prod_{k'=1}^{a'} \prod_{j'=1}^{a'} \rho'_{j'k'} \rightarrow K'_{j'k'}$

So we have

$$(E \triangleright S_1) \cap (\neg E \triangleright S_2)$$

$$= \prod_{k=1}^{a} (E \triangleright \prod_{j=1}^{a} (\rho_{jk} \rightarrow K_{jk})) \cap \prod_{k'=1}^{a'} (\neg E \triangleright \prod_{j'=1}^{a'} (\rho'_{j'k'} \rightarrow K'_{j'k'}))$$

$$= \prod_{k=1}^{a} ((E \triangleright (\rho_{1k} \rightarrow K_{1k})) \triangleright \prod_{j=2}^{a} (\rho_{jk} \rightarrow K_{jk})) \cap \prod_{k'=1}^{a'} (\neg E \triangleright (\rho'_{1k'} \rightarrow K'_{1k'})) \triangleright \prod_{j'=2}^{a'} (\rho'_{j'k'} \rightarrow K'_{j'k'})))$$

$$= \prod_{k=1}^{a} ((E \triangleright K_{1k})) \triangleright \prod_{j=2}^{a} (\rho_{jk} \rightarrow K_{jk})) \cap \prod_{k'=1}^{a'} (\neg E \triangleright K'_{1k'}) \triangleright \prod_{j'=2}^{a'} (\rho'_{j'k'} \rightarrow K'_{j'k'})))$$

$$= \prod_{k=1}^{a} ((\rho_{1k} \rightarrow (E \triangleright K_{1k})) \triangleright \prod_{j=2}^{a} (\rho_{jk} \rightarrow K_{jk})) \cap \prod_{k'=1}^{a'} (\rho'_{1k'} \rightarrow (\neg E \triangleright K'_{1k'})) \triangleright \prod_{j'=2}^{a'} (\rho'_{j'k'} \rightarrow K'_{j'k'})))$$

$$= \prod_{k=1}^{a} ((\rho_{1k} \rightarrow (E \triangleright (E_{1k} \triangleright T_{1k}))) \triangleright \prod_{j=2}^{a} (\rho_{jk} \rightarrow K_{jk})) \cap \prod_{k'=1}^{a'} (\rho'_{1k'} \rightarrow (\neg E \triangleright (E'_{1k'} \triangleright T'_{1k'}))) \triangleright \prod_{j'=2}^{a'} (\rho'_{j'k'} \rightarrow K'_{j'k'})))$$

$$= \prod_{k=1}^{a} ((\rho_{1k} \rightarrow (E \land E_{1k} \triangleright T_{1k})) \triangleright \prod_{j=2}^{a} (\rho_{jk} \rightarrow K_{jk})) \cap \prod_{k'=1}^{a'} (\rho'_{1k'} \rightarrow (\neg E \land K'_{1k'}))) \triangleright \prod_{j'=2}^{a'} (\rho'_{j'k'} \rightarrow K'_{j'k'})))$$
\[ E'_{1,k'} \triangleright T'_{1,k'} \triangleright \prod_{j'=2}^{n}(\rho_{j'k'} \rightarrow K^*_{j'k'}) \]

by the associativity of \((\cdot \triangleright \cdot)\)

\[ (\bigcap_{k=1}^{s} \prod_{j=1}^{n}(\rho_{jk} \rightarrow K^*_{jk})) \cap (\bigcap_{k'=1}^{s'} \prod_{j'=1}^{n'}(\rho_{j'k'} \rightarrow K^*_{j'k'})) \]

by the associativity of \((\cdot \cap \cdot)\)

\[ \prod_{k''=1}^{s''} \prod_{j''=1}^{n''} (\rho_{j''k''} \rightarrow K^*_{j''k''}) \]

If \(j = 1\), \(K^*_{jk} = (E \land E_1k) \triangleright T_{1k}\). Otherwise for any \(j > 1\), \(K^*_{jk} = K_{jk}\). Similarly, if \(j' = 1\), \(K'^*_{j'k'} = (\neg E \land E'_{1k'}) \triangleright T_{1k'}\). Otherwise for any \(j' > 1\), \(K'^*_{j'k'} = K_{j'k'}\).

For any \(k'' \leq s\), \(\rho_{j''k''} = \rho_{j''k''} = K^*_{j''k''} = n'' = n\), and for any \(k'' > s\), \(\rho_{j''k''} = \rho_{j''(k''-a)} = K^*_{j''(k''-a)}\) and \(n'' = n'\).

(c) if \(F = (F_1 \triangleright F_2)\), then \(\alpha_S(F_1) = \text{true} = \alpha_S(F_2)\). So by induction assumption, we have \(F_1 = \bigcap_{k=1}^{s} \prod_{j=1}^{n} (\rho_{jk} \rightarrow K_{jk})\) and \(F_2 = \bigcap_{k'=1}^{s'} \prod_{j'=1}^{n'} (\rho_{j'k'} \rightarrow K^*_{j'k'})\).

Therefore we have \(F = F_1 \triangleright F_2\)

\[ = (\bigcap_{k=1}^{s} \prod_{j=1}^{n} (\rho_{jk} \rightarrow K_{jk})) \cap (\bigcap_{k'=1}^{s'} \prod_{j'=1}^{n'} (\rho_{j'k'} \rightarrow K^*_{j'k'})) \]

\[ = \bigcap_{k''=1}^{s''} \bigcap_{j''=1}^{n''} \rho_{j''k''} \rightarrow K^*_{j''k''} \]

Let \(p_1 = (k'' \mod s')\) and \(p_2 = (k'' \mod s)\). Thus, if \(j \leq n\), \(\rho_{j''k''} = \rho_{j''p_1}\) and \(K^*_{j''k''} = K_{j''p_1}\). Otherwise, for any \(j > n\), \(\rho_{j''k''} = \rho_{j''(n-p_1)}\) and \(K^*_{j''k''} = (K_{j''p_1} \triangleright K^*_{j''(n-p_2)})\).

(d) otherwise, if \(F = \text{for } F_1 \text{ from } F_2 \text{ to } F_3 \text{ do } F_4\), then by the assumption(6), \(F\) can be reduced to a plain form. According to the above proof, we have \(F = \bigcap_{k=1}^{s} \prod_{j=1}^{n} (\rho_{jk} \rightarrow K_{jk})\).

3. Finally,

(a) if \(F = \text{early } F_1\), \(F = \text{if } F_1 \text{ then } F_2 \text{ else } F_3\), or \(F = F_1 \triangleright F_2\), we have \(\alpha(F) = \alpha_S(F)\). Thus \(F\) is reducible to the form \(\bigcap_{k=1}^{s} \prod_{j=1}^{n} K_{jk}\).

(b) if \(F = F_1 \parallel F_2\), then \(\alpha(F_1) = \alpha(F_2) = \text{true}\). By induction assumption,
$F_1 = \bigoplus_{l=1}^{t} \bigcap_{k=1}^{s} \bigcap_{j=1}^{n} K_{jkl}$ and $F_2 = \bigoplus_{l'=1}^{t'} \bigcap_{k'=1}^{s'} \bigcap_{j'_{1}=1}^{n'}$, thus we have

$$F = F_1 || F_2$$

$$= (\bigoplus_{l=1}^{t} \bigcap_{k=1}^{s} \bigcap_{j=1}^{n} \rho_{jkl} \rightarrow K_{jkl}) || (\bigoplus_{l'=1}^{t'} \bigcap_{k'=1}^{s'} \bigcap_{j'_{1}=1}^{n'} \rho_{j'_{1}}' \rightarrow K_{j'_{1}}')$$

$$= \bigoplus_{l=1}^{t+t'} \bigcap_{k''=1}^{s''} \bigcap_{j''_{1}=1}^{n''} \rho_{j''_{1}}'' \rightarrow K_{j''_{1}}''$$

For any $l'' \leq t$, we have $K''_{j''_{1}}''=K_{j''_{1}}''$, $\rho''_{j''_{1}}'=\rho_{j''_{1}}'$, $s''=s$ and $n''=n$; otherwise, for any $l'' > t$, we have $K''_{j''_{1}}''=K'_{j''_{1}}''(u-t)$, $\rho''_{j''_{1}}''=\rho_{j''_{1}}''(u-t)$, $s''=s'$ and $n''=n'$.

(c) otherwise, if $F = \text{par} F_1$ from $F_2$ to $F_3$ do $F_4$, then by the assumption(5), $F$ can be reduced to a plain form. According to the above proof, we have $F = \bigoplus_{l=1}^{t} \bigcap_{k=1}^{s} \bigcap_{j=1}^{n} \rho_{jkl} \rightarrow K_{jkl}$.

The proof is completed. □

**Corollary 13** Any well-formed specification in the basic LOGS can be reduced to the above form.

**Proof.** By Theorem 1, any well-formed expression $P$ in that basic LOGS can be represented as the following form.

$$P = \bigoplus_{l=1}^{t} \bigcap_{k=1}^{s} \bigcap_{j=1}^{n} E_{jkl} \rightarrow (\text{early} \bigotimes_{i=1}^{m} \text{after} X_{ijkl} = E_{ijkl})$$

Thus, by the assumption(5) in Law 3, the above form can be represented as follows.

$$P = \bigoplus_{l=1}^{t} \bigcap_{k=1}^{s} \bigcap_{j=1}^{n} \emptyset \rightarrow E_{jkl} \rightarrow (\text{early} \bigotimes_{i=1}^{m} \text{after} X_{ijkl} = E_{ijkl})$$

The proof is completed. □

According to Corollary 13, the normal form 5.2 is more expressive, and able to present all the well-formed programs in that basic LOGS.
5.4 Detecting communication interference

In order to avoid undesirable communication interference, this step intends to analyze the potential conflict. Before investigating the validation of communication interference, we define the set of auxiliary variables $\mathcal{X}_1$. These variables do not directly appear in the language, but are generated as by-products.

$$\mathcal{X}_1 = \{ x \cdot t | x \in \mathcal{X}, t \in \text{RANGE} \}$$

As a subset of $\mathcal{X}$, these variables is composed of primitive variables and range types.

The interference detection is based on the relation $\bowtie$. Primitive variables have interference iff they are identical. For the variables in $\mathcal{X}_1$, the relation is defined as follows.

$$x_1 \bowtie R_1 \bowtie x_2 \bowtie R_2 = (x_1 \bowtie x_2) \wedge (R_1 \bowtie R_2)$$

The relation $\bowtie$ detects whether two range types overlap. For instance, Range(1,3) $\bowtie$ Range(2,4) is true. For the variables in $\mathcal{X}_1$, the interference occurs iff the contained primitive variables are identical and their range types overlap.

We introduce the function $\beta_0 : (\mathcal{X} \times (\mathcal{X}_0 \rightarrow \text{TYPE})) \rightarrow (\mathcal{X}_0 \cup \mathcal{X}_1)$, which operates on variables and produces their corresponding primitive ones in either $\mathcal{X}_0$ or $\mathcal{X}_1$.

$$\beta_0(X, \rho) = \begin{cases} x \cdot \epsilon^\ast(E, \rho) & (X = (x \#E)) \\ x & \text{otherwise} \end{cases}$$

The function $\epsilon^\ast$ calculates the type for a given expression under a certain environment. If element selection is applied on the function, a variable in $\mathcal{X}_1$ will be produced. Otherwise, $\beta_0$ directly returns a primitive variable.

The function $\beta^* : (P \cup I) \times (\mathcal{X}_0 \rightarrow \text{TYPE}) \rightarrow P(\mathcal{X})$ is quite similar to $\beta$, but involves environments and accepts the instructions: par and for. The operator $\rightarrow$ updates the environment.
**Def 19** \( \beta^*(P, \rho) = \bigcup_{k=n_1}^{n_2} \beta^*(S, \rho\{x \mapsto \text{Range}(k,k)}) \)

\[
\begin{align*}
\beta^*(P_1 || P_2, \rho) &= \beta^*(P_1, \rho) \cup \beta^*(P_2, \rho) \\
\beta^*(S_1 ; S_2, \rho) &= \beta^*(S_1, \rho) \\
\beta^*(\text{Fr}, \rho) &= \beta^*(S, \rho\{x \mapsto \text{Range}(n_1,n_1)}) \\
\beta^*(\text{if } E \text{ then } S_1 \text{ else } S_2, \rho) &= \beta^*(S_1 \cap S_2, \rho) = \beta^*(S_1, \rho) \cup \beta^*(S_2, \rho) \\
\beta^*(\rho \rightarrow S, \rho') &= \beta^*(S, \rho') \\
\beta^*(E \Rightarrow S, \rho) &= \beta^*(S, \rho) \\
\beta^*(I_1 \odot I_2, \rho) &= \beta^*(I_1, \rho) \cup \beta^*(I_2, \rho) \\
\beta^*(\text{early } I, \rho) &= \beta^*(I, \rho) \\
\beta^*(\text{after } X = E, \rho) &= \{\beta_0(X, \rho)\}
\end{align*}
\]

where \( n_1 \) and \( n_2 \) represent the evaluation results of \( E_1 \) and \( E_2 \) under the environment \( \rho \).

**Law 3** requires that both \( \text{par} \) and \( \text{for} \) are reducible to plain forms. In details, under the function \( \beta^* \), they have the following properties.

**Law 4 (Additional assumptions for \( \beta^* \))**

\[
\begin{align*}
(5)^* \text{ par } x \text{ from } E_1 \text{ to } E_2 \text{ do } S &= \bigcup_{k=n_1}^{n_2} ((x \mapsto \text{Range}(k,k)) \rightarrow S) \\
(6)^* \text{ for } x \text{ from } E_1 \text{ to } E_2 \text{ do } S &= ((x \mapsto \text{Range}(n_1,n_1)) \rightarrow S)
\end{align*}
\]

where \( n_1 \) and \( n_2 \) represents the values of \( E_1 \) and \( E_2 \) under a certain environment.

**Lemma 3** *The abstract function \( \beta^* \) satisfies the laws in Law 4.*

**Proof.**

1. Need to show \( \beta^*(\text{par } x \text{ from } E_1 \text{ to } E_2 \text{ do } S, \rho) = \beta^*(\bigcup_{k=n_1}^{n_2} \{x \mapsto \text{Range}(k,k)\} \rightarrow S), \rho) \).

\[
\begin{align*}
\beta^*(\text{par } x \text{ from } E_1 \text{ to } E_2 \text{ do } S, \rho) \\
= & \bigcup_{k=n_1}^{n_2} \beta^*(S, \rho\{x \mapsto \text{Range}(k,k)}) \\
= & \bigcup_{k=n_1}^{n_2} \beta^*(\{x \mapsto \text{Range}(k,k)\} \rightarrow S, \rho)
\end{align*}
\]

by \( \beta \) definition

\[
\begin{align*}
&= \bigcup_{k=n_1}^{n_2} \beta^*(\{x \mapsto \text{Range}(k,k)\} \rightarrow S, \rho)
\end{align*}
\]
$\beta^* (\prod_{k=1}^{n_1} \{ x \mapsto \text{Range}(k,k) \} \to S, \rho)$

2. Need to show $\beta^*(\text{for } x \text{ from } E_1 \text{ to } E_2 \text{ do } S, \rho) = \beta^*(\{ x \mapsto \text{Range}(n_1,n_1) \} \to S, \rho)$.

$\beta^*(\text{for } x \text{ from } E_1 \text{ to } E_2 \text{ do } S, \rho)$

$= \beta^*(S, \rho \upharpoonright \{ x \mapsto \text{Range}(n_1,n_1) \})$

$= \beta^*(\{ x \mapsto \text{Range}(n_1,n_1) \} \to S, \rho)$

The proof is completed.

**Proposition 14** The abstract function $\beta^*$ satisfies all the laws in Law 1 and Law 3.

**Proof.**

1. Need to show that the function $\beta^*$ satisfies all the laws in Law 1. The proof can be completed like Proposition 2.

2. Need to show that the function $\beta^*$ satisfies all the laws in Law 3.

(a) Need to show $\beta^*(\rho_1 \rightarrow (\rho_2 \rightarrow S), \rho) = \beta^*((\rho_1 \uparrow \rho_2) \rightarrow S, \rho)$.

$\beta^*(\rho_1 \rightarrow (\rho_2 \rightarrow S), \rho)$

$= \beta^*(S, \rho \uparrow \rho_1 \uparrow \rho_2)$

$= \beta^*((\rho_1 \uparrow \rho_2) \rightarrow S, \rho)$
(b) Need to show $\beta^*(E \triangleright (\rho \rightarrow S), \rho') = \beta^*(\rho \rightarrow (E \triangleright S), \rho')$.

\[
\beta^*(E \triangleright (\rho \rightarrow S), \rho') \\
= \beta^*(S, \rho' \vdash \rho) \\
= \beta^*(E \triangleright S, \rho' \vdash \rho) \\
= \beta^*(\rho \rightarrow (E \triangleright S), \rho')
\]

by definition

by definition

by definition

by definition

by definition

by definition

(c) Need to show $\beta^*(\rho \rightarrow (S_1 ; S_2), \rho') = \beta^*((\rho \rightarrow S_1) ; S_2, \rho')$.

\[
\beta^*(\rho \rightarrow (S_1 ; S_2), \rho') \\
= \beta^*(S_1, \rho' \vdash \rho) \\
= \beta^*((\rho \rightarrow S_1), \rho') \\
= \beta^*((\rho \rightarrow S_1) ; S_2, \rho')
\]

by definition

by definition

by definition

by definition

by definition

by definition

(d) Need to show $\beta^*(\emptyset \rightarrow S, \rho) = \beta^*(S, \rho)$. It is obvious.

(e) Need to show par $x$ from $E_1$ to $E_2$ do $S$ can be reduced to a plain form. Shown from Lemma 3.

(f) Need to show for $x$ from $E_1$ to $E_2$ do $S$ can be reduced to a plain form. Shown from Lemma 3.

Therefore the function $\beta^*$ satisfies all the laws. □
Lemma 4 \ A well-formed specification $P$ contains communication interference in its first superstep iff $\beta^*(P, \rho) = X$ under the environment $\rho$.

Proof. By Proposition 14 and Theorem 12, we apply the normal form (5.2) on this function. For any $P$, if there exist $i_1, k_1, l_1$ and $i_2, k_2, l_2$ such that $i_1 \neq i_2$ or $l_1 \neq l_2$ and $\beta_0(X_{i_1k_1l_1}, \rho) \ll \beta_0(X_{i_2k_2l_2}, \rho)$, then $\beta^*(P, \rho) = X$; otherwise, $\beta^*(P, \rho) = \{\beta_0(X_{i_1k_1l_1}, \rho) \mid l \leq t, k \leq s, i \leq m\}$. The proof is completed. \hfill \Box

Corollary 15 \ For any well-defined specification $P$ in the basic LOGS, $\beta^*(P, \rho) = \beta(P)$ under the environment $\rho$.

Proof. By Theorem 1, we apply the normal form (4.3) on $\beta$, thus gain $\{x_{i_1k_1l_1} \mid l \leq t, k \leq s, i \leq m\}$. The form is also applied on the function $\beta^*$, thus $\beta^*(P, \rho) = \{\beta_0(x_{i_1k_1l_1}, \rho) \mid l \leq t, k \leq s, i \leq m\}$. By its definition, $\beta_0(x, \rho) = x$. Therefore these functions produce a identical result. \hfill \Box

This corollary presents that for the instructions in the basic LOGS, the functions produces the same result under a certain environment. With the function $\beta^*$, we are also able to properly check the interference for the programs in that basic LOGS.

Similar to $\gamma$, the function $\gamma^*: \mathcal{P} \times (\mathcal{X}_0 \rightarrow \text{TYPE}) \rightarrow \mathcal{P} \cup \{\Pi\}$ returns the tail for a given specification. The operator $\rightarrow$ is able to update the environment. The instruction $\text{par}$ takes the same effect as a number of parallel processes. Similarly, $\text{for}$ can be represented as a sequence of commands, each of which has an individual value of the index variable.
Def 20 \[ \gamma^*(P_1 \parallel P_2, \rho) = \gamma^*(P_1, \rho) \parallel \gamma^*(P_2, \rho) \quad (\gamma^*(P_1, \rho) \neq \Pi, \gamma^*(P_2, \rho) \neq \Pi) \]
\[ \gamma^*(P_1 \parallel P_2, \rho) = \gamma^*(P_1, \rho) \quad (\gamma^*(P_2, \rho) = \Pi) \]
\[ \gamma^*(P_1 \parallel P_2, \rho) = \gamma^*(P_2, \rho) \quad (\gamma^*(P_1, \rho) = \Pi) \]
\[ \gamma^*(S_1 \uplus S_2, \rho) = \gamma^*(S_1, \rho) \uplus S_2 \quad (\gamma^*(S_1, \rho) \neq \Pi) \]
\[ \gamma^*(S_1 \uplus S_2, \rho) = S_2 \quad (\gamma^*(S_1, \rho) = \Pi) \]
\[ \gamma^*(\text{if } E \text{ then } S_1 \text{ else } S_2, \rho) = \gamma^*(S_1 \sqcap S_2, \rho) \]
\[ \gamma^*(S_1 \sqcap S_2, \rho) = \gamma^*(S_1, \rho) \sqcap \gamma^*(S_2, \rho) \quad (\gamma^*(S_1, \rho) \neq \Pi, \gamma^*(S_2, \rho) \neq \Pi) \]
\[ \gamma^*(S_1 \sqcap S_2, \rho) = \gamma^*(S_1, \rho) \quad (\gamma^*(S_2, \rho) = \Pi) \]
\[ \gamma^*(S_1 \sqcap S_2, \rho) = \gamma^*(S_2, \rho) \quad (\gamma^*(S_1, \rho) = \Pi) \]
\[ \gamma^*(\rho \rightarrow S, \rho') = \gamma^*(S, \rho' \parallel \rho) \]
\[ \gamma^*(E \rightarrow S, \rho) = \gamma^*(S, \rho) \]
\[ \gamma^*(\text{early } I, \rho) = \Pi \]
\[ \gamma^*(\Pi, \rho) = \Pi \]

Def 21 \[ \gamma^*(F_r, \rho) = \gamma^*(\prod_k^{n_0} = n_1 \{x \mapsto \text{Range}(k, k) \rightarrow S\}, \rho) \]
\[ \gamma^*(F_r, \rho) = \gamma^*(\prod_k^{n_2} = n_1 \{x \mapsto \text{Range}(k, n_2) \rightarrow S\}, \rho) \]

Law 5 (Additional assumptions for \( \gamma \))

\[(5)^* \text{par } x \text{ from } E_1 \text{ to } E_2 \text{ do } S = \prod_k^{n_0} = n_1 \{x \mapsto \text{Range}(k, k) \rightarrow S\}\]
\[(6)^* \text{for } x \text{ from } E_1 \text{ to } E_2 \text{ do } S = \prod_k^{n_2} = n_1 \{x \mapsto \text{Range}(k, n_2) \rightarrow S\}\]

Lemma 5 The abstract function \( \gamma^* \) satisfies the laws in Law 5.

Proof. Shown from its definition. 

Proposition 16 The abstract function \( \gamma^* \) satisfies all the laws in Law 1 and Law 3.

Proof. Similar to Proposition 14 and Shown from its definition and Lemma 5.
Lemma 6  The abstract function $\gamma^*$ returns the tail of a given well-formed specification under a certain environment.

Proof. By Proposition 16 and Theorem 12, we apply the normal form (5.2) on this function. Thus, for any well-formed specification $P$, we have $\gamma^*(P, \rho) = \prod_{i=1}^{n} \prod_{k=1: n(s,t)>1} \prod_{j=2}^{n} \rho_{ijkl} \rightarrow K_{ijkl}$. The proof is completed. □

Corollary 17  For a well-defined program $P$ in the basic LOGS, $\gamma^*(P, \rho) = \gamma(P)$ under the environment $\rho$.

Proof. By Theorem 1, we apply the normal form (5.2) on this function $\gamma^*$, thus gain

$\gamma^*(P, \rho) = \prod_{i=1}^{n} \prod_{k=1: n(s,t)>1} \prod_{j=2}^{n} E_{ijkl} \triangleright ( \text{early } \bigotimes_{i=1}^{m} \text{after } x_{ijkl} = E_{ijkl} ).$ The form is also applied on the function $\gamma^*$, thus we have

$\gamma^*(P, \rho) = \prod_{i=1}^{n} \prod_{k=1: n(s,t)>1} \prod_{j=2}^{n} E_{ijkl} \triangleright ( \text{early } \bigotimes_{i=1}^{m} \text{after } X_{ijkl} = E_{ijkl} ).$ For the basic LOGS, $X$ always represents a primitive variable $x$. So we have $\gamma^*(P, \rho) = \gamma(P).$ □

The abstract function $\delta^*: \mathcal{P} \cup \{ \Pi \} \times (\mathcal{X}_0 \rightarrow \text{TYPE}) \rightarrow \{ \text{true}, \text{false} \}$ uses the previous ones: $\beta^*$ and $\gamma^*$ to check the interference for a given specification superstep by superstep. It is formalized as $(P = \Pi) \land (\beta^*(P, \rho) \neq \mathcal{X}) \land (\delta^*(\gamma^*(P, \rho), \rho)).$

Proposition 18 The function $\delta^*$ satisfies all the laws in Law 1 and Law 3.

Proof. Shown from Proposition 14 and Proposition 16.

Theorem 19  A well-formed specification has communication interference iff the function $\delta^*$ returns false.

Proof. By Proposition 18, we apply the normal form (5.2) on this function, thus have

$(P = \Pi) \land (\beta^*(P, \rho) \neq \mathcal{X}) \land (\delta^*(\gamma^*(P, \rho), \rho))$

= by definition
The formulation \( \{ \beta_0(X_{s t}, \rho) \mid l \leq t, k \leq s, i \leq m \} \neq \mathcal{X} \) checks communication interference in the first superstep for the specification \( P \). The term \( \delta^*(\prod_{l=1}^{\ell} \prod_{k=1}^{s} \prod_{j=1}^{n} O_{jkl}, \rho) \) recursively checks the interference for the rest part of this given program. Therefore, the conflict detection of \( P \) becomes to verify the inference from its first superstep to the last one. If one of them returns false, this whole specification has communication conflict. \( \square \)

**Corollary 20** For a well-defined program \( P \) in the basic LOGS, \( \delta^*(P, \rho) = \delta^*(P) \) under the environment \( \rho \).

**Proof.** Shown from Theorem 19, Corollary 15 and Corollary 17. \( \square \)

### 5.5 Type checking

Before investigating type checking for the extended language, we first introduce the new data type - array.

\[
\text{ARRAY} = \{ \text{Array}(T, n) \mid T \in \{\text{Int}, \text{Bool}\}, n \in \mathbb{N} \}
\]

This type is composed of two ingredients: element type and array length. We define a couple of array-based functions. The function \( \text{elm} : \text{ARRAY} \rightarrow \{\text{Int}, \text{Bool}\} \) returns the element type for a given array. The other one \( \text{len} : \text{ARRAY} \rightarrow \text{RANGE} \) gives the length of an array. Note that it represents the length by a range type rather than an integer.

To involve the environment \( \rho \), we define the function \( \epsilon^* \) based on \( \epsilon \). Aside from environment involvement, the former has quite similar behaviors as the latter, but allows to compute the type for element selection.
The function $e^*$ possesses strong type checking rules. It is able to not only detect type inconsistency for arithmetical and logical calculation, but statically approximate the unsafe access to array elements. For instance, let us define an integer array $A$ with 2 elements. The expression $e^*(A\#3,\rho)$ returns Error to indicate the out-of-index access.

**Lemma 7** For a well-defined program $P$ in the basic LOGS, $e^*(P,\rho) = \varepsilon(P)$ under the environment $\rho$.

**Proof.** Obvious. □

The abstract function $\xi : S \times (\chi_0 \rightarrow \text{TYPE}) \rightarrow (\chi_0 \rightarrow \text{TYPE})$ captures environment modification and return an updated environment.

**Def 24**

$\xi(S_1 ; S_2, \rho) = \xi(S_2 , \xi(S_1 , \rho))$

$\xi(\rho \rightarrow S, \rho') = \xi(S , \rho'\upharpoonright\rho))$

$\xi(S, \rho) = \rho \quad \text{otherwise}$

For sequential composition, $\xi$ recursively catches environment change. Only the operator $\rightarrow$ is able to change the environment. For other instructions, the environment $\rho$ keeps unchanged.

**Def 25**
\[ \varepsilon^*(P_1 \parallel P_2, \rho) = \varepsilon^*(P_1, \rho) \land \varepsilon^*(P_2, \rho) \land \varepsilon^*(S_1 ; S_2, \rho) = \varepsilon^*(S_1, \rho) \land \varepsilon^*(S_2, \xi(S_1, \rho)) \]

\[ \varepsilon^*(F_r, \rho) = (\tau(x) \leq \text{Int}) \land \{ \varepsilon^*(E_1, \rho), \varepsilon^*(E_2, \rho) \} \subseteq \text{RANGE}_0 \land \]

\[ \land_{k=n_1}^{n_2} \varepsilon^*(S, \rho \{ x \mapsto \text{Range}(k, k) \}) \]

\[ \varepsilon^*(P_{x_0}, \rho) = \varepsilon^*(E, \rho) \quad \varepsilon^*(S, \rho) = \varepsilon^*(S_1, \rho) \land \varepsilon^*(S_2, \rho) \quad \varepsilon^*(S, \rho) = \varepsilon^*(E, \rho) \land \varepsilon^*(S, \rho) \]

\[ \varepsilon^*(\text{after } x = E, \rho) = \varepsilon^*(E, \rho) \quad \varepsilon^*(\text{after } x \# S_1 = S_2, \rho) = \varepsilon^*(E_2, \rho) \land \varepsilon^*(E_1, \rho) \]

where \( n_1 \) and \( n_2 \) represents the values that the evaluation of \( E_1 \) and \( E_2 \) produce under the environment \( \rho \).

**Law 6 (Additional assumptions for \( \varepsilon^* \))**

1. \((5)^* \) par \( \{ x \text{ from } E_1 \text{ to } E_2 \ \text{do } S \} = \| n_2 \|_{n_1}(\{ x \mapsto \text{Range}(k, k) \} \rightarrow S) \)

2. \((6)^* \) for \( \{ x \text{ from } E_1 \text{ to } E_2 \ \text{do } S \} = \{ x \mapsto \text{Range}(n_1, n_2) \} \rightarrow S \)

where \( n_1 \) and \( n_2 \) represents the values that the evaluations of \( E_1 \) and \( E_2 \) produce under a certain environment.

**Lemma 8** The abstract function \( \varepsilon^* \) satisfies the laws in Law 6.

**Proof.**

1. Need to show \( \varepsilon^*(\{ x \text{ from } E_1 \text{ to } E_2 \ \text{do } S \}, \rho) = \varepsilon^*(\| n_2 \|_{n_1}(\{ x \mapsto \text{Range}(k, k) \} \rightarrow S), \rho) \).

\[ \varepsilon^*(\{ x \text{ from } E_1 \text{ to } E_2 \ \text{do } S \}, \rho) = \varepsilon^*(E_1, \rho) \land \varepsilon^*(E_2, \rho) \quad \text{by } \varepsilon^* \text{ definition} \]
(\tau(x) \sqsubseteq \text{Int}) \land \{\varepsilon^*(E_1, \rho), \varepsilon^*(E_2, \rho)\} \subseteq \text{RANGE}_0 \land
\bigwedge_{k=n_1}^{n_2} \varepsilon^*(S, \rho \upharpoonright \{x \mapsto \text{Range}(k, k)\})
= \text{by well-formness}
\bigwedge_{k=n_1}^{n_2} \varepsilon^*(S, \rho \upharpoonright \{x \mapsto \text{Range}(k, k)\})
= \bigwedge_{k=n_1}^{n_2} \varepsilon^*(\{x \mapsto \text{Range}(k, k)\} \rightarrow S, \rho)
= \varepsilon^*(\bigwedge_{k=n_1}^{n_2} \{x \mapsto \text{Range}(k, k)\} \rightarrow S, \rho)

2. Need to show \varepsilon^*(\text{for } x \text{ from } E_1 \text{ to } E_2 \text{ do } S, \rho) = \varepsilon^*(\{x \mapsto \text{Range}(n_1, n_2)\} \rightarrow S, \rho).
\varepsilon^*(\text{for } x \text{ from } E_1 \text{ to } E_2 \text{ do } S, \rho)
= \text{by } \varepsilon^* \text{ definition}
(\tau(x) \sqsubseteq \text{Int}) \land \{\varepsilon^*(E_1, \rho), \varepsilon^*(E_2, \rho)\} \subseteq \text{RANGE}_0 \land \varepsilon^*(S, \rho \upharpoonright \{x \mapsto \text{Range}(n_1, n_2)\})
= \varepsilon^*(S, \rho \upharpoonright \{x \mapsto \text{Range}(n_1, n_2)\})
= \varepsilon^*(\{x \mapsto \text{Range}(n_1, n_2)\} \rightarrow S, \rho)

Therefore the function \varepsilon^* satisfies all the laws. \qed

**Proposition 21** The abstract function \varepsilon^* satisfies all the laws in Law 1 and Law 3.

**Proof.**

1. Need to show that it satisfies all the laws in Law 1. Here only presents the proof for the law(5) in Law 1. Others can be easily completed.
\[ \epsilon^*(E \triangleright (S_1 ; S_2), \rho) \]

\[ = \quad \text{by definition} \]

\[ \epsilon(E, \rho) \triangleright \text{Bool} \land \epsilon^*(S_1 ; S_2, \rho) \]

\[ = \quad \text{by definition} \]

\[ \epsilon(E, \rho) \triangleright \text{Bool} \land \epsilon^*(S_2, \xi(S_1, \rho)) \]

\[ = \quad \text{by definition} \]

\[ \epsilon(E, \rho) \triangleright \text{Bool} \land \epsilon^*(S_2, \xi(E \triangleright S_1, \rho)) \]

\[ = \quad \text{by definition} \]

\[ \epsilon^*((E \triangleright S_1); S_2, \rho) \]

2. Need to show that it satisfies all the laws in Law 3. Here only presents the proof for the law(3). The proof of the law(5) and the law(6) is shown from Law 21.

\[ \epsilon^*(\rho \triangleright (S_1 ; S_2), \rho') \]

\[ = \quad \text{by definition} \]

\[ \epsilon^*(S_1, \rho \triangleright \rho) \land \epsilon^*(S_2, \xi(S_1, \rho \triangleright \rho)) \]

\[ = \quad \text{by definition} \]

\[ \epsilon^*(\rho \triangleright S_1, \rho') \land \epsilon^*(S_2, \xi((\rho \triangleright S_1), \rho')) \]

\[ = \quad \text{by definition} \]

\[ \epsilon^*((\rho \triangleright S_1); S_2, \rho') \]

Therefore the function \( \epsilon^* \) satisfies all the laws. \( \square \)

**Corollary 22** For a well-defined program \( P \) in the basic LOGS, \( \epsilon^*(P, \rho) = \epsilon(P) \) under the environment \( \rho \).
Proof. By Theorem 1, a well-formed specification in the basic LOGS can be represented as follows.

\[ P = \prod_{i=1}^{t} \prod_{k=1}^{s} \prod_{j=1}^{n} E_{ijkl} \supset \left( \text{early } \bigotimes_{i=1}^{m} \text{ after } x_{ijkl} = E_{ijkl} \right) \]

We apply the above form on the function \( \varepsilon(P) \) under the environment \( \rho \), thus gain

\[ \wedge_{i=1}^{t} \wedge_{k=1}^{s} \wedge_{j=1}^{n} (\varepsilon(E_{ijkl}, \rho) \leq \text{Bool}) \wedge (\wedge_{i=1}^{m} \varepsilon(E_{ijkl}) \leq \tau(x_{ijkl})) \].

Then, we apply the program on the function \( \varepsilon^*(P, \rho) \). For the programs in the basic LOGS, all the instructions do not change the environment. Therefore the initial environment \( \rho \) distributes into the whole program. Thus we have \( \wedge_{i=1}^{t} \wedge_{k=1}^{s} \wedge_{j=1}^{n} (\varepsilon^*(E_{ijkl}, \rho) \leq \text{Bool}) \wedge (\wedge_{i=1}^{m} \varepsilon^*(E_{ijkl}) \leq \tau(x_{ijkl})) \). By Lemma 7, we have \( \varepsilon^*(E, \rho) = \varepsilon(E) \). Therefore for any program in the basic LOGS, the functions produce identical results under the environment \( \rho \).

5.6 Variable registration for communication

The abstract function \( \omega \) collects the variables that need to be registered for a given expression. Its definition on element selection is presented as follows.

**Def 26** \( \omega(\text{before } x \# E) = \{x\} \)

The function \( \zeta \) produces the set of variables to be registered. It collects the variables that appear in \( \text{after } X \) and \( \text{before } X \). Its definitions on the additional instructions are shown as follows.

**Def 27**

\[ \zeta(\Pr) = \zeta(\Fr) = \zeta(\rho \rightarrow S) = \zeta(S) \]

**Proposition 23** The abstract function \( \zeta \) satisfies all the laws in Law 1 and Law 3.

**Proof.** Shown from the definition of \( \zeta \).
Theorem 24  The set of all variables in after $X$ and before $X$ of a well-formed specification $P$ is $\zeta(P)$.

Proof. By Proposition 23 and Theorem 12, we apply the normal form 5.2 on this function. Thus, for any $P$, we have $\zeta(P) = \bigcup_{ijkl} (\omega(E_{ijkl}) \cup \sigma(X_{ijkl}) \cup \omega(E_{ijkl}))$ where $\sigma(x) = \sigma(x\#E) = x$. □

5.7 Analysis of communication dependencies

We define the function $u^* : \mathcal{E} \times (X_0 \rightarrow \text{TYPE}) \rightarrow (X_0 \cup X_1)$ to collect the variables that communicate with other processes for a given expression. In the other word, $\omega^*$ collects all the variables following before. Note that the generated set may contain the variables in both $X_0$ and $X_1$. For the basic LOGS, the functions: $u^*$ and $\omega$ always produce the same result.

Def 28  $\omega^*(E_1 + E_2, \rho) = \omega^*(E_1, \rho) \cup \omega^*(E_2, \rho)$

$\omega^*(E_1 \land E_2, \rho) = \omega^*(E_1, \rho) \cup \omega^*(E_2, \rho)$

$\omega^*(\text{before } X, \rho) = \{\beta_0(X, \rho)\}$

$\omega^*(-E, \rho) = \omega^*(E, \rho)$

$\omega^*(v, \rho) = \omega^*(v, \rho) = \{\}$

The function $\theta$ accounts the number of sequential processes for a specification. We only show the definition for $\Pr$. Clearly, the value of $\theta(\Pr)$ is 1.

Def 29  $\theta(\Pr) = (\text{val } \epsilon(E_2)) - (\text{val } \epsilon(E_1)) + 1$

Like $\eta$, the function $\eta^* : \mathbb{N} \times \mathbb{N} \times (\mathcal{P} \cup \mathcal{I}) \times (X_0 \rightarrow \text{TYPE}) \rightarrow \mathbb{P}(X)$ is defined to produce the set of variables to be read under the environment $\rho$. 
Def 30 $\eta^*(i, n, p_x, \rho) = \eta^*(i, i, S, \rho \{x \mapsto \text{Range}(i-n, i-n)\})$ \hspace{1cm} (n \leq i < n + \theta(Fx))

$\eta^*(i, n, P_1 \parallel P_2, \rho) = \eta^*(i, n, P_1, \rho) \cup \eta^*(i, n+\theta^*(P_1, \rho), P_2, \rho)$

$\eta^*(i, n, S, \rho) = \{} \hspace{1cm} (i \neq n)$

$\eta^*(i, n, Fx, \rho) = \eta^*(i, n, S, \rho \{x \mapsto \text{Range}(n_1, n_2)\})$ \hspace{1cm} (i = n)

$\eta^*(i, n, S_1 ; S_2, \rho) = \eta^*(i, n, S_1, \rho) \cup \eta^*(i, n, S_2, \xi(S_1, \rho))$ \hspace{1cm} (i = n)

$\eta^*(i, n, \text{if } E \text{ then } S_1 \text{ else } S_2, \rho) = \eta^*(i, n, E, \rho) \cup \eta^*(i, n, S_1 \cap S_2, \rho)$ \hspace{1cm} (i = n)

$\eta^*(i, n, S_1 \cap S_2, \rho) = \eta^*(i, n, S_1, \rho) \cup \eta^*(i, n, S_2, \rho)$ \hspace{1cm} (i = n)

$\eta^*(i, n, \rho \rightarrow S, \rho') = \eta^*(i, n, S, \rho' \rho)$ \hspace{1cm} (i = n)

$\eta^*(i, n, E \triangleright S, \rho) = \eta^*(i, n, E, \rho) \cup \eta^*(i, n, S, \rho)$ \hspace{1cm} (i = n)

$\eta^*(i, n, \text{early } I, \rho) = \eta^*(i, n, I, \rho)$ \hspace{1cm} (i = n)

$\eta^*(i, n, I_1 \otimes I_2, \rho) = \eta^*(i, n, I_1, \rho) \cup \eta^*(i, n, I_2, \rho)$ \hspace{1cm} (i = n)

$\eta^*(i, n, \text{after } X = E, \rho) = \omega^*(E, \rho)$ \hspace{1cm} (i = n)

Law 7 (Additional assumptions for $\eta^*$)

(5)* par $x$ from $E_1$ to $E_2$ do $S = \|n_2\|_{k = n_1}(\{x \mapsto \text{Range}(k, k)\} \rightarrow S)$

(6)* for $x$ from $E_1$ to $E_2$ do $S = \{x \mapsto \text{Range}(n_1, n_2)\} \rightarrow S$

where $n_1$ and $n_2$ represents the values that the evaluation of $E_1$ and $E_2$ produce under the environment $\rho$.

Lemma 9 The abstract function $\eta^*$ satisfies the laws in Law 7.

1. Need to show $\eta^*(i, n, \text{par } x \text{ from } E_1 \text{ to } E_2 \text{ do } S, \rho) = \eta^*(i, n,$

$\|n_2\|_{k = n_1}(\{x \mapsto \text{Range}(k, k)\} \rightarrow S), \rho)$.

$\eta^*(i, n, \|n_2\|_{k = n_1}(\{x \mapsto \text{Range}(k, k)\} \rightarrow S), \rho)$

$= \text{processes share common variables, } n \leq i < n + \theta^*(Fx, \rho)$

$\eta^*(i, i, \{x \mapsto \text{Range}(i-n, i-n)\} \rightarrow S, \rho)$

$= $
\[ \eta^*(i, i, S, \rho \uparrow \{x \mapsto \text{Range}(i - n, i - n)\}) \]

\[ = \]

\[ \eta^*(i, n, \text{par } x \text{ from } E_1 \text{ to } E_2 \text{ do } S, \rho) \]

2. Need to show \( \eta^*(i, n, \text{for } x \text{ from } E_1 \text{ to } E_2 \text{ do } S, \rho) = \eta^*(i, n, \{x \mapsto \text{Range}(n_1, n_2)\} \rightarrow S, \rho) \).

\[ \eta^*(i, n, \text{for } x \text{ from } E_1 \text{ to } E_2 \text{ do } S, \rho) \]

\[ = \]

\[ \eta^*(i, n, S, \rho \uparrow \{x \mapsto \text{Range}(n_1, n_2)\}) \]

\[ = \]

\[ \eta^*(i, n, \{x \mapsto \text{Range}(n_1, n_2)\} \rightarrow S, \rho) \]

**Proposition 25** The function \( \eta^* \) satisfies all the laws in Law 1 and Law 3.

**Proof.**

1. Need to show that the function \( \eta^* \) satisfies all the laws in Law 1. Here only presents the proof for the law(5) in Law 1. Others can be easily stated.

\[ \eta^*(i, n, E \triangleright (S_1 ; S_2), \rho) \]

\[ = \]

\[ \eta^*(i, n, E, \rho) \cup \eta^*(i, n, S_1 ; S_2, \rho) \]

\[ = \]

\[ \eta^*(i, n, E, \rho) \cup \eta^*(i, n, S_1, \rho) \cup \eta^*(i, n, S_2, \xi(S_1, \rho)) \]

\[ = \]

\[ \eta^*(i, n, E \triangleright S_1, \rho) \cup \eta^*(i, n, S_2, \xi(S_1, \rho)) \]

\[ = \]
\[ \eta^*(i, n, E \triangleright S_1, \rho) \cup \eta^*(i, n, S_2, \xi(E \triangleright S_1, \rho)) \]

= 

\[ \eta^*(i, n, (E \triangleright S_1) ; S_2, \rho) \]

2. Need to show that the function \( \eta^* \) satisfies all the laws in Law 3. By Lemma 9, both \( \text{Pr} \) and \( \text{Fr} \) can be reduced to plain forms. Here only shows the proof for the law(3).

\[ \eta^*(i, n, \rho \rightarrow (S_1 ; S_2), \rho') \]

= by definition

\[ \eta^*(i, n, \rho \rightarrow (S_1 ; S_2), \rho'\uparrow\rho) \]

= by definition

\[ \eta^*(i, n, S_1, \rho'\uparrow\rho) \cup \eta^*(i, n, S_2, \xi(S_1, \rho'\uparrow\rho)) \]

= by definition

\[ \eta^*(i, n, (\rho \rightarrow S_1), \rho') \cup \eta^*(i, n, S_2, \xi((\rho \rightarrow S_1), \rho')) \]

= by definition

\[ \eta^*(i, n, (\rho \rightarrow S_1) ; S_2, \rho') \]

Therefore the function satisfies all the laws. □

**Theorem 26** The abstract function \( \eta^*(i, n, P, \rho) \) collects the set of all variables read by process \( i \), a sequential process in a well-formed specification \( P \) with starting process id \( n \).

**Proof.** By Proposition 25 and Theorem 12, we apply the normal form (5.2) on this function. Thus, for any \( P \), if \( n' \leq i' < n' + t \), let \( l \equiv i' - n' + 1 \), and then \( \eta^*(i', n', P, \rho) = \bigcup_{ijk} (\omega^*(E_{ijkl}, \rho) \cup \omega^*(E_{ijkl}, \rho)) \); otherwise, \( \eta^*(i', n', P, \rho) = \{ \} \). □
Corollary 27 For any well-formed program P in the basic LOGS, \( \eta^*(i, n, P, \rho) = \eta(i, n, P) \) under the environment \( \rho \).

Proof. Shown from Theorem 11 and Theorem 26. \( \square \)

5.8 Generating code for expressions

Similar to \( \phi \), the abstract function \( \phi^* : \mathcal{E} \times (\mathcal{X}_0 \rightarrow \text{TYPE}) \rightarrow \text{String} \) translate a LOGS expression into its corresponding C expression as a string. For any expression \( E \), if \( e^*(E, \rho) \) produce a range type in RANGE\(_0\), its bound will be printed out. Otherwise, the following functions defines its output.

**Def 31**

\[
\begin{align*}
\phi^*(E_1 + E_2, \rho) &= \phi^*(E_1, \rho) + "+" + \phi^*(E_2, \rho) \\
\phi^*(E_1 \land E_2, \rho) &= \phi^*(E_1, \rho) + "\land" + \phi^*(E_2, \rho) \\
\phi^*(E_1 \# E_2, \rho) &= \phi^*(E_1, \rho) + "\#" + \phi^*(E_2, \rho)"I" \\
\phi^!(E, \rho) &= "!" + \phi^*(E, \rho) \\
\phi^*(\text{before } x, \rho) &= \phi^*(x, \rho) = "x" \\
\phi^*(v, \rho) &= "v".
\end{align*}
\]

This approach benefits the transformation from not only simplifying the target code, but also eliminating some temporary variables (e.g., index variable in PR) in LOGS, but not in the target language.

5.9 Generating code for communications

The function \( \lambda^* \) collects the numbers of process id that read the given variable \( X \). Note that, unlike \( \lambda \), this function is also able to approximate the communication for element selection. It can be formalized as follows.

\[
\lambda^*(X, P, \rho) \equiv \{ i < \theta^*(P, \rho) \mid \exists X' \in \eta^*(i, 0, P, \rho) \cdot X \bowtie X' \}
\]
For a shared variable, its modification may influence other processes. The abstract function
\[ \psi^* : (S_0 \cup I) \times P \times (X_0 \rightarrow \text{TYPE}) \rightarrow \text{String} \]
of communication code generation needs to broadcast all the processes that need to know the change. So the function takes the whole program as one of its arguments.

**Def 32**

\[
\psi^*(\text{Fr}, P, \rho) = \text{"for (int } x + \psi_0(x) + \text{"} + \psi^*(E_1, P, \rho) + \text{"}; } +
\psi_0(x) + \text{"<"} + \psi^*(E_1, P, \rho) + \text{"}; } + \psi_0(x++) + \text{"} \} \{ \text{'}
\psi^*(S, P, \rho \{x \rightarrow \text{Range}(n_1, n_2)\})
\text{'} \}
\]

\[
\psi^*(\text{if } E \text{ then } S_1 \text{ else } S_2, P, \rho) = \text{"if"} + \phi^*(E, P) + \text{"{"} } + \psi^*(S_1, P, \rho) + \text{"} + \text{" else" } + \psi^*(S_2, P, \rho) + \text{"} \} \]

\[
\psi^*(S_1; S_2, P, \rho) = \psi^*(S_1, P, \rho) + \psi^*(S_2, P, \xi(S_1, \rho))
\]

\[
\psi^*(\text{early } I, P, \rho) = \psi^*(I, P, \rho) + \text{'bsp__sync();'}
\]

\[
\psi^*(I_1 \otimes I_2, P, \rho) = \psi^*(I_1, P, \rho) + \psi^*(I_2, P, \rho)
\]

\[
\psi^*(\text{after } x = E, P, \rho) = \text{"x :=" } + \phi^*(E, P) + \text{";" } +
\int_{i : \lambda^*(x, P, \rho)} \text{"bsp__put } (i, x, 0, \text{sizeof } (\tau(x)); \text{" )" } \]

\[
\psi^*(\text{after } x \neq E_1 = E_2, P, \rho) = \phi^*(x \neq E_1, \rho) + \text{":=" } + \phi^*(E_2, P) + \text{";" } + \int_{i : \lambda^*(x \neq E_1, \rho), P, \rho} \text{"bsp__put } (i, x, \phi^*(E_1, \rho), \text{sizeof } (\text{elm } \tau(x)); \text{" )" }
\]

### 5.10 Generating code for processes

Similar to \( \kappa \), the abstract function
\[ \kappa^* : (\mathbb{N} \times \mathcal{P} \times \mathcal{P} \times (X_0 \rightarrow \text{TYPE})) \rightarrow \text{String} \]
generates the code for a given program under a certain environment. A parallel process is decomposed of a number of sequential processes. Each sequential process starts with printing the registration of shared variables, then its body and ending with generating the code for releasing these registered variables.

**Def 33**
\[ \kappa^*(n, P_1, P, \rho) = \sum_{k=1}^{n_2} \kappa^*(n, S, P, \rho \{ x \mapsto \text{Range}(k, k) \}) \]
\[ \kappa^*(n, P_1 \parallel P_2, P, \rho) = \kappa^*(n, P_1, P, \rho) + \kappa^*(n + \theta^*(P_1, \rho), P_2, P, \rho) \]
\[ \kappa^*(n, S, P, \rho) = \text{void process n \{ " + } \]
\[ \int_{x: \xi(P)} \text"bsp.pushregister(\&x, sizeof(\tau(x))\});" + \]
\[ \psi^*(S, P, \rho) + \]
\[ \int_{x: \xi(P)} \text"bsp.popregister(\&x);" + \]
\[ "\}" \]

For the program \( P \) that succeeds passing syntactical checking, communication interference checking and type checking, its target C code can be produced as the following form:

\[ \text"#include \ldots" + \kappa^*(0, P, P, \rho) + \text"void main() \ldots" \]

where we assume that \( \rho = \tau \).

### 5.11 Summary

This chapter extends the previous basic LOGS to include advanced instructions including par, for and element section. Correspondingly, the original normal form is updated to present more expressive programs. The most significant difference comes from environment involvement. To collapse to the normal form, both par and for must be reducible to plain forms. The primitive language can be generally considered as a sub-language of the advanced one. So it is able to present all the well-formed programs in the basic LOGS.

Since array index may not be determined statically, the renewed abstract function \( \beta^* \) approximates communication interference. For two array accesses, if their ranges overlap, the interference assumedly occurs. The language provides a restricted finite iteration, which tail is the new iteration with an increased lower bound. Code generation may print either an expression as a string in the target language or a concrete value. The printing style depends on whether the value of this expression...
can be statically determined. The approach benefits the translation from not only simplifying target code, but also eliminating some expressions existing in LOGS, but not in target language.

The following factors make FLEXIBO more suitable for this kinds of application. Firstly, the unusual flexible language eliminates many syntactical restrictions that commonly exist in many languages, therefore enables developers to present abstract and expressive specifications. By comparison, most widely used languages cannot reserve such capability. Secondly, FLEXIBO enables developers to define desirable constraints. Unlike FLEXIBO, LOGS has many restrictions in both syntax and semantics. How to present these desirable constraints is the key requirement to freely write LOGS programs. In this case, we define a small sub-language with more restricted syntax based on FLEXIBO. This redefinition is straightforward and natural, does not force developers to involve in the complicated process for either lexical or syntactical checking. Thirdly, the mechanism reflection is presented through a pure object-oriented style. Object-oriented programming possesses great reusability and extendability. FLEXIBO enables developers to override the behavior of program evaluation with object-oriented manner. It reserves inheritance relation in any reflection system. Therefore, developers only need to redefine the instructions expected to be overridden, others are able to reuse the behavior through inheritance. Finally, FLEXIBO is able to guarantee both flexibility and preformation. LOGS-based programs may be highly abstract to express various parallel programs. They will not be executed in LOGS or FLEXIBO platform. On the contrary, the specifications are transformed to BSPLib-linked C code, then efficiently run in that target environment.
Chapter 6

FLEXIBO's Formal Semantics

6.1 Introduction

Various mathematical models [38, 42, 92] have been proposed to give clear and concise description for language implementation. Denotational semantics [27, 69] defines a set of functions [70] to map programs to mathematical objects (denotations). The reasoning of language's properties becomes to study these denotations [68]. A fundamental term in this technique is compositionality. It defines an expression, even a program through its constituent parts. However, denotational semantics does not straightforwardly show the details of language implementation [79]. On the contrary, operational semantics [4, 5] aims to directly model the process of program execution. In particular, it shows how to interpret a valid program as a sequence of computational steps. The most common approach to define operational semantics is to use state transition systems for program evaluation. This technique has been widely used to formally study program properties such as the relations between processes (e.g., bisimulation [66, 67]).

In general, there are two different ways to define operational semantics. One is called natural semantics [39, 47]. It describes how to evaluate an expression in one step under a certain state. Its transition relation specifies the relationship between the initial state and the final state, does not show any intermediate steps. For instance, if we ignore evaluation state, the natural semantics of
binary condition can be simplified as follows.

$$
\begin{align*}
E_1 \rightarrow \text{true} & \quad E_2 \rightarrow v \\
\text{if } E_1 \text{ then } E_2 \text{ else } E_3 \rightarrow v
\end{align*}
$$

$$
\begin{align*}
E_1 \rightarrow \text{false} & \quad E_3 \rightarrow v \\
\text{if } E_1 \text{ then } E_2 \text{ else } E_3 \rightarrow v
\end{align*}
$$

The other is structural operational semantics (SOS)[77, 78]. It emphasizes on each concrete step of program execution. SOS can present how to evaluate an expression step by step. Accordingly, the SOS-style semantics for binary condition is shown as follows.

$$
\begin{align*}
E_1 \rightarrow E_4 & \\
\text{if } E_1 \text{ then } E_2 \text{ else } E_3 \rightarrow \text{if } E_4 \text{ then } E_2 \text{ else } E_3
\end{align*}
$$

$$
\begin{align*}
\text{if } \text{true} \text{ then } E_2 \text{ else } E_3 \rightarrow E_2 \\
\text{if } \text{false} \text{ then } E_2 \text{ else } E_3 \rightarrow E_3
\end{align*}
$$

To give a clear presentation for FLEXIBO’s implementation, this chapter tries to define natural semantics for FLEXIBO’s program execution. Note that the semantics is not concerned about variable and attribute declaration. These issues are assumed to be properly completed before program execution. Variables and attributes are represented as ID numbers and name strings respectively. For instance, a temporary variable $x$ may correspond to an integer (e.g., 1) as its ID.

In the semantics, each step of program reduction needs an environment, which stores a series of global values and a temporary store. The environment can be modeled as a directed graph with a single entry point. The global store records all the values during program execution. The temporary store is dynamically allocated for computation, and released after its completion. For instance, method invocation needs some spaces for local variables. These are allocated before method invocation, and automatically freed after its execution.

The semantics operates on a set of reduction rules, each of which defines how to translate a configuration paired with an expression and an environment to another configuration paired with a value (a vertex in environment graph) and an updated environment. Each reduction step evaluates the expression under that environment, produces a value and update that environment for further
computation. The semantics does not specially handle statements, but generally regards them as expressions. For instance, this language allows developers to present expressive programs such as 

\[ x := (\text{if } (x == 0) 1 \text{ else } 2). \]

Rather than fully exhibiting complicated formal definitions, this chapter incrementally defines the semantics for FLEXIBO. At the beginning, it shows the semantics for imperative languages. The simplest one even does not include the instructions like loop. In order to enhance expressivity, extended languages provide more advanced instructions such as complicated data structure-array. After properly defining these semantics, this chapter introduces a small object-oriented language based on them. With a few extra instructions, this semantics is able to show object-oriented features.

### 6.2 Environment and its graph representation

Program evaluation must be under an environment, which mainly includes two parts: global store and running stack. The environment can be abstractly modeled as a directed graph with an entry point, which starts traversing the graph.

#### 6.2.1 Graph representation

A graph is a set of vertices and the set of edges that connect pairs of vertices.

\[
\begin{align*}
\mathcal{P} & \quad \text{Set of graphs} & \mathcal{V} & \quad \text{Set of vertices} \\
\mathcal{G} & \quad \text{Set of edges} & \mathcal{L} & \quad \text{Set of labels}
\end{align*}
\]

A directed edge corresponds to a triplet of initial node, label and terminal node. An initial node may have multiple edges to reach a terminal node, but they are not allowed to share common labels. In a graph, a head(initial node) maps to a set of mapping from labels to tail nodes. In general, there are three kinds of vertices, each of which have different properties. For instance, some of them stand for the root of that graph, another for the nodes that end graph traversing, the final one for the rest.
Entry point is a special vertex to start traversing a graph. Each environment graph has a unique root (see Figure 6.1). The function \( \text{root} : \mathcal{P} \rightarrow \mathcal{V} \) returns the point for a certain graph. A few predefined edges originate from this node. In general, they correspond to three kinds of labels.

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>this, cenv, excp</td>
</tr>
<tr>
<td>2</td>
<td>stack</td>
</tr>
<tr>
<td>3</td>
<td>constant (e.g., 1, true, null)</td>
</tr>
</tbody>
</table>

Deadend is the final vertex that terminates graph traversing. In the other word, there is no vertex that starts from deadend. Each deadend may be labeled with a constant string such as "1" or "true".

Diagram 6.1 shows all kinds of vertices and edges. The top vertex is the root node to traverse the whole graph. Several predefined edges stem from there. The later section explains what they stand for. A few edges labeled with constants start with the root and end with the deadend. The diagram also shows an object Tom with two edges, which link to the deadend: 30 and true respectively.

**Figure 6.1: Graph**

With these notions, a graph \( \rho : \mathcal{V} \rightarrow (\mathcal{L} \rightarrow \mathcal{V}) \) has the following functions to extract its properties. The function \( \text{vSet} : \mathcal{P} \rightarrow \mathcal{P}(\mathcal{V}) \) returns the set of vertices for a graph. \( \text{edge} : \mathcal{P} \times \mathcal{V} \rightarrow (\mathcal{L} \rightarrow \mathcal{V}) \) gives the instructions constructed by labels and tail vertices for some head vertex.

- \( \text{root} : \mathcal{P} \rightarrow \mathcal{V} \) return the root for a graph
- \( \text{vSet} : \mathcal{P} \rightarrow \mathcal{P}(\mathcal{V}) \) return the set of vertices
- \( \text{edge} : \mathcal{P} \times \mathcal{V} \rightarrow (\mathcal{L} \rightarrow \mathcal{V}) \) return edges for a given vertex in a graph

The function tail return the tail vertex for a graph. If the initial vertex and label are known, the corresponding terminal vertex can be uniquely determined. The function relink modifies the link...
to a new vertex, therefore returns a updated graph. The function clone makes a copy for some vertex, and returns the pair of its updated graph and the new vertex. clone(ρ, v) = ρ · v' where \( vSet(\rho') = vSet(\rho) \cup \{v'\} \) and edge(ρ', v') = edge(ρ, v).

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>tail</td>
<td>return the terminal vertex</td>
</tr>
<tr>
<td>relink</td>
<td>relink the reference to a vertex</td>
</tr>
<tr>
<td>clone</td>
<td>clone a vertex</td>
</tr>
</tbody>
</table>

Diagram 6.2 illustrates how to modify the link to a vertex. Initially, the edge age from Tom links to 30. After the operation, it is relinked to 31.

![Figure 6.2: Relink](image)

### 6.2.2 Primitive value, Object and Array

FLEXIBO's primitive values directly correspond to the deadend, which has no references to other vertices. The following table shows how to represent a deadend under a certain environment. For instance, true stands for the deadend linked by an edge labeled true. Other vertices map to FLEXIBO's usual objects. They may contain a set of references to other objects. The vertices are considered as the locations to store objects. Similarly, labels in the graph corresponds to attributes directly. Further discussion does not identify the differences between deadend and primitive values, vertices and objects, labels and attributes, graph and state.
### 6.3: Array

Errors and arrays are also first-class values in FLEXIBO. Each error is identified by a string. Let $\mathcal{R}$ and $\mathcal{Y}$ denote the set of FLEXIBO’s errors and arrays respectively. Thus we have

$$
\mathcal{R} = \{ \text{err } \overline{s} \ | \ \overline{s} \in \mathcal{S} \}
$$

$$
\mathcal{Y} = \{ \text{ary } (v_1, \ldots, v_n) \ | \ v_i \in \mathcal{V} \}
$$

where $\mathcal{S}$ stands for the set of FLEXIBO’s constant strings. An array can be considered as an object, and consists of a sequence of values. Its indices are the labels to refer to its components. Figure 6.3 shows that a Boolean array $v$ has three elements. The first one links to false, the rest to true.

Array supports the computation-extension. Two arrays can be merged through the operator $\oplus$. For instance, let $y_1$ and $y_2$ denote $\text{ary } (v_1, \ldots, v_n)$ and $\text{ary } (v'_1, \ldots, v'_n)$ respectively, then we have $y_1 \oplus y_2 = \text{ary } (v_1, \ldots, v_n, v'_1, \ldots, v'_n)$. Diagram 6.4 shows the array-based computation. The arrays: $v_1$ and $v_2$ have one element and two elements respectively. Thus the produced array $v_3$ contains all the three elements. Its first one links to the same vertex as the element in $v_1$. Similarly, its second and third elements share the common vertices with the corresponding ones in $v_2$.

The boolean functions: $\text{isErrV}$, $\text{isMtd}$, $\text{isCls}$, $\text{isAry}$ check the explicit type for a certain value. If the given value is an error, the first one returns true. Similarly, the second one makes a check for a method, others for a class and an array respectively.
FLEXIBO evaluation environment needs several global attributes, each of which refers to a global value. For instance, the edge labeled by this links to the value of this. Similarly, cenv and excp stand for the labels of the edges to the current running environment and the current exception. Diagram 6.1 shows these attributes. The value of this points to the object Tom, which has two references such as name and married to the corresponding deadend. currentEnv shares a common object with this. The current exception and the temporary stack in the graph are assigned as null.

<table>
<thead>
<tr>
<th>this</th>
<th>reference to this</th>
</tr>
</thead>
<tbody>
<tr>
<td>cenv</td>
<td>reference to current environment</td>
</tr>
<tr>
<td>excp</td>
<td>reference to the thrown exception</td>
</tr>
<tr>
<td>stack</td>
<td>reference to a temporary stack</td>
</tr>
</tbody>
</table>

The environment provides a number of functions to define this semantics. In general, their role can be categorized as store-based and error-based. We introduce these functions and show their details as follows.

### 6.2.3 Store-based

The environment manages the change of this and currentEnv, which stand for the current object and the current running environment respectively. For instance, evaluating the expression o.1 needs
to change the value of currentEnv. The attribute 1 will be evaluated under the value of o. The following functions are defined to manage these issues.

\[ \text{setThis} : \mathcal{P} \times \mathcal{V} \rightarrow \mathcal{P} \]
\[ \text{getThis} : \mathcal{P} \rightarrow \mathcal{V} \]
\[ \text{setCntEnv} : \mathcal{P} \times \mathcal{V} \rightarrow \mathcal{P} \]
\[ \text{getCntEnv} : \mathcal{P} \rightarrow \mathcal{V} \]

where \( \mathcal{P} \) and \( \mathcal{V} \) denote the sets of environments and values.

The first two functions show this-related operations. The function setThis is used to change the value of this. The function getThis returns the object that this points to. Their mathematical definitions are shown as follows.

\[ \text{setThis}(p, v) = \text{relin}(p, \text{root } p, \text{this}, v) \]
\[ \text{getThis } p = \text{tail}(p, \text{root } p, \text{this}) \]

Similarly, we can present the formal definitions for setCntEnv and getCntEnv.

\[ \text{setCntEnv}(p, v) = \text{relin}(p, \text{root } p, \text{cenv}, v) \]
\[ \text{getCntEnv } p = \text{tail}(p, \text{root } p, \text{cenv}) \]

To access to the attributes of objects needs the following functions.

\[ \text{setVal} : \mathcal{P} \times \mathcal{L} \times \mathcal{V} \rightarrow \mathcal{P} \]
\[ \text{getVal} : \mathcal{P} \times \mathcal{L} \rightarrow \mathcal{P} \]

The functions: setVal and getVal are to modify and obtain the value of a given attribute respectively. They are formally defined as follows.

\[ \text{setVal}(p, l, v) = \text{relin}(p, \text{tail}(p, \text{root } p, \text{cenv}), l, v) \]
\[ \text{getVal}(p, l) = \text{tail}(p, \text{tail}(p, \text{root } p, \text{cenv}), l) \]

Temporary store is allocated before method invocation or class initialization. The following functions are designed to take operations on the temporary store.

\[ \text{setStack} : \mathcal{P} \times \mathcal{V} \rightarrow \mathcal{P} \]
\[ \text{getStack} : \mathcal{P} \rightarrow \mathcal{V} \]
Their mathematical definitions are shown as follows.

\[
\text{setStack}(\rho, v) = \text{relink}(\rho, \text{root } \rho, \text{stack, } v) \\
\text{getStack } \rho = \text{tail}(\rho, \text{root } \rho, \text{stack})
\]

To modify and access to temporary variables, we define the following functions.

\[
\text{setValStack} : \mathcal{P} \times \mathcal{L} \times \mathcal{V} \to \mathcal{P} \\
\text{getValStack} : \mathcal{P} \times \mathcal{L} \to \mathcal{V}
\]

Accordingly, their formal definitions are shown as follows. A temporary variable corresponds to an ID. FLEXIBO regards the ID as a special attribute in the temporary store.

\[
\text{setValStack}(\rho, x, v) = \text{relink}(\rho, \text{tail}(\rho, \text{root } \rho, \text{stack}), x, v) \\
\text{getValStack}(\rho, x) = \text{tail}(\rho, \text{tail}(\rho, \text{root } \rho, \text{stack}), x)
\]

### 6.2.4 Error-based

The environment defines the following functions to handle errors.

\[
\text{isErr} : \mathcal{P} \to \{ \text{true, false} \} \\
\text{setErr} : \mathcal{P} \times \mathcal{R} \to \mathcal{P} \\
\text{getErr} : \mathcal{P} \to \mathcal{R} \\
\text{match} : \mathcal{P} \times \mathcal{S} \to \{ \text{true, false} \}
\]

where \(\mathcal{P}\) and \(\mathcal{R}\) denote the sets of environments and errors.

The first three functions are to check, set and get an error from a given environment. Since an error is a first-class value in this language, it cannot be a direct indicator to cease programming evaluation. The information can be obtained from the environment. An error thrown will be set into the environment. Therefore the function \text{isErr} returns true to stop execution. Mathematically, these functions can be defined as follows,

\[
\text{isErr } \rho = (\text{tail}(\rho, \text{root } \rho, \text{excp}) \neq \text{null}) \\
\text{setErr}(\rho, r) = \text{relink}(\rho, \text{root } \rho, \text{excp}, r) \\
\text{getErr } \rho = \text{tail}(\rho, \text{root } \rho, \text{excp})
\]
If the vertex `excp` does not link to `null`, the function `isErr` returns `true`. The function `setErr` modifies the edge labeled with `excp` to link to a new value. The function `getErr` directly returns the value that `excp` links to. The function `match` compares the formal string with that string identifying the active error. For most errors, this operation checks whether the formal is a substring of the latter.

The evaluation of the expressions (e.g., `break`) triggers instruction errors, which cannot be caught by developers, but the language. Checking on these errors with `isErr` always produces `false`. The following functions are particularly defined to handle these cases.

\[
\begin{align*}
\text{isBrkErr} : & \mathcal{P} \to \{ \text{true, false} \} \\
\text{isCtnErr} : & \mathcal{P} \to \{ \text{true, false} \} \\
\text{isRtnErr} : & \mathcal{P} \to \{ \text{true, false} \} \\
\text{setBrkErr} : & \mathcal{P} \to \mathcal{P} \\
\text{setCtnErr} : & \mathcal{P} \to \mathcal{P} \\
\text{setRtnErr} : & \mathcal{P} \times \mathcal{V} \to \mathcal{P} \\
\text{getRtnV} : & \mathcal{R} \to \mathcal{V}
\end{align*}
\]

The first three functions above check the explicit types (e.g., produced by `continue`) of instruction errors. The second three ones are to set the errors into environments. Note that the error generated by `return` contains an value for a returned value for method invocation. The function `getRtnV` extracts the contained value from `return` error. A normal value applied on the function `getRtnV` returns itself. Here ignores their mathematical definitions, which can be similarly obtained from the previous ones.

### 6.3 Imperative languages

This section defines a set of imperative languages and their formal semantics. It begins with the simplest language with fundamental instructions. A language supporting loop extends from this primitive one. A further extended language supports composite data structure-array.

The semantics operate on the structure-configuration. It mathematical definition is shown as
follows. In general, there are two kinds of configuration. One is paired with an expression and its evaluation environment. The other is constructed by a value (a vertex in environment graph) and a updated environment. The set $\mathcal{E}$ stands for FLEXIBO's expressions. All FLEXIBO programs including variables and values are in the set $\mathcal{E}$.

\[
\text{CONT} = \{(E, \rho) \mid E \in \mathcal{E}, \rho \in \mathcal{P}\}
\]

\[
\overline{\text{CONT}} = \{(v, \rho) \mid v \in \mathcal{V}, \rho \in \mathcal{P}\}
\]

Reduction rules are defined in the following form. Any well-formed configuration in the set $\text{CONT}$ can be reduced to a corresponding configuration in $\overline{\text{CONT}}$.

\[
\rightsquigarrow : \text{CONT} \rightarrow \overline{\text{CONT}}
\]

Unlike the semantics in [56], the configuration term $\overline{\text{CONT}}$ encodes a updated environment. The aim is to clearly show the change on environments during program evaluation.

### 6.3.1 A primitive language $\mathcal{L}_P$

The first imperative language only has several fundamental instructions. Its operational semantics is given to formally illustrate how to evaluate the instructions. The syntax of this language $\mathcal{L}_P$ is shown as follows.

\[
E = E ; E \mid \text{if } E \text{ then } E \text{ else } E \mid E := E \mid x \mid T
\]

- $E ; E$ sequential composition
- if $E$ then $E$ else $E$ binary condition
- $E := E$ assignment
- $x$ variable
- $T$ constant eg. 1, true

We begin with reduction rules of sequential composition. This following rules present that the expression $E_1 ; E_2$ firstly evaluates the expression $E_1$. If the evaluation does not generate an error, the second expression $E_2$ will be evaluated.

\[
(E_1, \rho) \rightsquigarrow (v, \rho') \quad \text{isErr } \rho'
\]

\[
(E_1 ; E_2, \rho) \rightsquigarrow (v, \rho')
\]
The function \( \text{isErr} \) is used to check whether the evaluation of the expression \( E_1 \) triggers an error. Note that we cannot directly check from the value of \( E_1 \) since an error is also a value, only an error thrown stops evaluating.

The following rules show configuration reduction for the instruction-binary condition.

\[
\frac{(E_1, \rho) \leadsto (v_1, \rho') \quad \text{isErr} \quad (E_2, \rho') \leadsto (v_2, \rho'')}{(E_1 \; ; \; E_2, \rho) \leadsto (v_2, \rho'')} \]

If evaluating \( E_1 \) generates \( \text{true} \), the expression \( E_2 \) will be evaluated. Conversely, it evaluates the expression \( E_3 \) if \( \text{false} \) is produced. If the evaluation of the condition triggers an error, none of branches will be evaluated. When this expression reduces to a non-Boolean value, an error is thrown to indicate the unexpected result.

\[
\frac{(E_1, \rho) \leadsto (\text{true}, \rho') \quad (E_2, \rho') \leadsto (v, \rho'')}{(\text{if } E_1 \text{ then } E_2 \text{ else } E_3, \rho) \leadsto (v, \rho'')}
\]

\[
\frac{(E_1, \rho) \leadsto (\text{false}, \rho') \quad (E_3, \rho') \leadsto (v, \rho'')}{(\text{if } E_1 \text{ then } E_2 \text{ else } E_3, \rho) \leadsto (v, \rho'')}
\]

To define reduction rules for assignment, we introduce an auxiliary expression

\[\mathcal{U} = \{ E \leftarrow v \mid E \in \mathcal{E}, v \in \mathcal{V} \}\]

This notion presents a physical update for the expression \( E \). Its semantics on a temporary variable is given as follows.

\[
(x \leftarrow v, \rho) \leadsto (\text{null}, \text{setValStack}(\rho, x, v))
\]

This rule assumes that the variable exists in the temporary store, and is properly bound to a vertex.

With the auxiliary notion above, the semantics of assignment is defined as follows.

\[
\frac{(E_2, \rho) \leadsto (v_2, \rho') \quad \text{isErr} \quad (E_1 \leftarrow v, \rho') \leadsto (v', \rho'')}{(E_1 := E_2, \rho) \leadsto (\text{null}, \rho'')}
\]
\[(E_2, \rho) \leadsto (v, \rho') \text{ isErr } \rho'] \\
(E_1 := E_2, \rho) \leadsto (v, \rho')

If the expression \(E_2\) does not reduce to any error, \(E_1\) is updated as \(v\) with the previous rule. On the contrary, this produced error is directly returned as the result of this whole expression.

If a temporary variable is properly bound, its value can be directly obtained through the function getValStack.

\[(x, \rho) \leadsto \text{getValStack}(\rho, x, \rho)\]

Evaluating FLEXIBO's constant expression returns a corresponding deadend directly. Let capital letters stand for the constant expressions, their corresponding small letters for the edges to link to deadend.

\[(T, \rho) \leadsto (\tilde{t}, \rho)\]

6.3.2 Extended imperative language \(L_X\)

The previous subsection defines a simple imperative language without loop or other advanced instructions. This part extends that language to be more powerful and expressive with some additional instructions. In details, the extended language has the following syntax. \(E_p\) stands for the instructions appear in the previous subsection.

\[E = \text{skip} \mid \text{throw } E \mid \text{while } E \ E \mid \text{return } E \mid \text{continue} \mid \text{break} \mid \text{try } E \ \text{catch } S \ E \mid E_p\]

<table>
<thead>
<tr>
<th>Skip</th>
<th>Do nothing</th>
<th>Continue</th>
<th>Continue command</th>
</tr>
</thead>
<tbody>
<tr>
<td>Throw (E)</td>
<td>Throw an error</td>
<td>Break</td>
<td>Break command</td>
</tr>
<tr>
<td>While (E) (E)</td>
<td>Loop</td>
<td>Try (E) \text{ catch } S \ E</td>
<td>&quot;catch&quot; instruction</td>
</tr>
<tr>
<td>Return (E)</td>
<td>Return a value</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The expression skip does nothing, therefore does not change the environment.

\[(\text{skip}, \rho) \leadsto (\text{null}, \rho)\]
There exist a number of reduction rules for throw. The first one says that this error will be returned directly if the evaluation in $E$ yields an error. The second rule presents that if the evaluation does not trigger an error and the value produced is an error, it will be set into the environment. The final one describes that if the value generated is not an error, a default error is thrown to indicate this undesired value.

$$(E, \rho) \rightsquigarrow (v, \rho') \Rightarrow (\text{isErr } \rho \text{'} \text{, Err} \rho \text{'}$$

$$(E, \rho) \rightsquigarrow (v, \rho') \Rightarrow (\text{isErr } \rho \text{'} \Rightarrow (\text{isErr } \rho \text{'} = \text{Err } v)

$$(E, \rho) \rightsquigarrow (v, \rho') \Rightarrow (\text{isErr } \rho \text{'} \Rightarrow (\text{isErr } \rho \text{'} \Rightarrow (\text{Err } v, \text{setErr}(\rho, \text{v}))

$$(E, \rho) \rightsquigarrow (v, \rho') \Rightarrow (\text{isErr } \rho \text{'} \Rightarrow (\text{isErr } \rho \text{'} \Rightarrow (\text{Err } v, \text{setErr}(\rho, \text{v}))

The formulation of while is more complicated. The first rule formalizes the case where evaluating $E_1$ produces an error. The second one shows the case that the condition reduces to false. The third rule formalizes that if evaluating the condition produces true, the expression $E_2$ is executed firstly followed by the loop again.

$$(E_1, \rho) \rightsquigarrow (v, \rho') \Rightarrow (\text{isErr } \rho \text{'} \Rightarrow (\text{while } E_1 E_2, \rho \Rightarrow (v, \rho'))

$$(E_1, \rho) \rightsquigarrow (\text{false}, \rho') \Rightarrow (\text{while } E_1 E_2, \rho \Rightarrow (\text{null}, \rho'))

$$(E_1, \rho) \rightsquigarrow (\text{true}, \rho') \Rightarrow (E_2; \text{while } E_1 E_2, \rho' \Rightarrow (v, \rho'') \Rightarrow (\text{isErrv} \Rightarrow (\text{while } E_1 E_2, \rho \Rightarrow (v, \rho''))

This language defines three kinds of instruction errors, which are triggered by the interpreter. The loop instruction is able to catch the errors thrown by break and continue and remove the errors from the environment.

$$(E_1, \rho) \rightsquigarrow (\text{true}, \rho') \Rightarrow (E_2, \rho'') \Rightarrow (v, \rho'') \Rightarrow (\text{isBrkErr} \rho'' \Rightarrow (\text{while } E_1 E_2, \rho \Rightarrow (\text{null}, \text{setErr}(\rho'', \text{null}))

$$(E_1, \rho) \rightsquigarrow (\text{true}, \rho') \Rightarrow (E_2; \text{while } E_1 E_2, \rho' \Rightarrow (v, \rho'' \Rightarrow (\text{isCtntErr} \rho'' \Rightarrow (\text{while } E_1 E_2, \rho \Rightarrow (\text{null}, \text{setErr}(\rho'', \text{null}))

$$(E_1, \rho) \rightsquigarrow (\text{true}, \rho') \Rightarrow (E_2, \rho'') \Rightarrow (v, \rho'' \Rightarrow (\text{isErr} \rho'' \Rightarrow (\text{isBrkErr} \rho'' \Rightarrow (\text{isCtntErr} \rho'')) \Rightarrow (\text{while } E_1 E_2, \rho \Rightarrow (v, \rho''))
For other errors, this instruction stop evaluating and returns that triggered error.

The evaluation of the instructions such as break, continue and return yields various predefined errors.

\[ \rho' = \text{setBrkErr}\rho \]
\[ (\text{break}, \rho) \sim (\text{getErr}\rho', \rho') \]
\[ \rho' = \text{setCtnuErr}\rho \]
\[ (\text{continue}, \rho) \sim (\text{getErr}\rho', \rho') \]

For the instruction return \( E \), the following rules say that this error will be directly returned if \( E \) reduces to an error, otherwise the interpreter wraps the value that \( E \) reduces to as a special error. This error is set into the environment to terminate further evaluation. It can be caught by method invocation. This wrapped value is considered as the returned value of the executed method.

\[ (E, \rho) \sim (v, \rho') \quad \text{isErr}\rho' \]
\[ (\text{return } E, \rho) \sim (v, \rho') \]

\[ (E, \rho) \sim (v, \rho') \quad \neg \text{isErr}\rho' \quad \rho'' = \text{setRtnErr}(\rho', v) \]
\[ (\text{return } E, \rho) \sim (\text{getErr}\rho'', \rho'') \]

The instruction \( \text{try } E_1 \text{ catch } S E_2 \) is formalized through the following rules. The first one presents the case where the evaluation of \( E_1 \) does not trigger an error.

\[ (E_1, \rho) \sim (v, \rho') \quad \neg \text{isErr}\rho' \]
\[ (\text{try } E_1 \text{ catch } S E_2, \rho) \sim (v, \rho') \]

For convenience, let \( P \) denote the predicate to check instruction errors.

\[ P(\rho) \equiv (\text{isRtnErr}\rho) \lor (\text{isBrkErr}\rho) \lor (\text{isCtnuErr}\rho) \]

The next one shows that this instruction cannot catch instruction errors such as break error.

\[ (E_1, \rho) \sim (v, \rho') \quad P(\rho') \]
\[ (\text{try } E_1 \text{ catch } S E_2, \rho) \sim (v, \rho') \]

If a normal error is thrown and the formal string does not match the error, the instruction returns the error directly. Otherwise, the expression \( E_2 \) will be evaluated. Note that FLEXIBO requires the
expression following catch must be a string expression. Its evaluation never changes the environment, and impossibly triggers an error.

\[
(E_1, \rho) \leadsto (v, \rho') \quad \text{isErr } \rho' \quad \neg \text{P}(\rho') \quad (S, \rho') \leadsto (S, \rho') \quad \text{match}(\rho', S) \\
(\text{try } E_1 \text{ catch } S \ E_2, \rho) \leadsto (v, \rho')
\]

\[
(E_1, \rho) \leadsto (v, \rho') \quad \text{isErr } \rho' \quad \neg \text{P}(\rho') \quad (S, \rho') \leadsto (S, \rho') \quad \text{match}(\rho', S) \quad \rho'' = \text{setErr}(\rho', \text{null}) \\
(E_2, \rho'') \leadsto (v', \rho''')
\]

(try \(E_1 \text{ catch } S \ E_2, \rho) \leadsto (v', \rho''')

The following expressions derives from the ones above. To evaluate an assertion is to check its logical expression. It can be represented as binary condition. Similarly, the expression-imply can be defined as the special case of binary condition.

\[
\text{assert } E \equiv \text{if } E \text{ then skip else throw error } "\text{Assertion is broken}" \\
E_1 \Rightarrow E_2 \equiv \text{if } E_1 \text{ then } E_2 \text{ else skip}
\]

6.3.3 Extended language \(L_A\) with array

The previous part extends the primitive language with more advanced expressions, but still lacks the support to composite data structure. Here defines the instruction-composite expression, which consists of a sequence of expressions. Its evaluation yields an array. The extended language has the following syntax. \(E_x\) stands for the syntax of \(L_X\).

\[
E = [E, \ldots, E] \mid E_x
\]

Let the notion \(E_c\) denote the set of composite expressions. There defines two functions for composite expressions. The first one \(\text{len} : E_c \rightarrow N\) returns the length for a given composite expression. The second function \(\text{elmAt} : E_c \times N \rightarrow E\) returns a component from a composite expression.

With the notions above, the following rule presents the semantics for composite expressions. If each component reduces to a value correctly, this expression yields an array composed of these values generated. Conversely, the error is returned as the value of the whole expression if a component
produces an error.

\[
(E_1, \rho_1) \rightsquigarrow (v_1, \rho_2) \quad \ldots \quad \neg \text{isErr} \rho_{n-1} \quad (E_n, \rho_{n-1}) \rightsquigarrow (v_n, \rho_n)
\]

\[
([E_1 \ldots E_n], \rho) \rightsquigarrow (\text{ary} [v_1 \ldots v_n], \rho_n)
\]

This language allows composite expressions to be assigned, but has the following restrictions.

1. The assigning value is an array.

2. They have the same length.

3. Each component accepts its corresponding value in that array.

\[
isAry v \quad \text{len}[E_1, \ldots, E_n] = \text{edge}(\rho, v) \quad (E_1 \leftarrow \text{tail}(\rho, v, \#1), \rho) \rightsquigarrow (v', \rho_1)
\]

\[
\ldots \neg \text{isErr}(\rho_{n-1}) \quad (E_n \leftarrow \text{tail}(\rho_{n-1}, v, \#n), \rho_{n-1}) \rightsquigarrow (v_n, \rho_n)
\]

\[
([E_1, \ldots, E_n] \leftarrow v, \rho) \rightsquigarrow (\text{null}, \rho_n)
\]

The operator \# is polymorphically used in this language. Depending on the value of \( E_1 \), this operator stands for various operations. If \( E_1 \) reduces to an array, this operator represents element selection. The following rules formalize element selection and update. The first one presents how to obtain an element from an array. If \( E_1 \) and \( E_2 \) reduces to an array and an index within the range of the array, this expression selects the value from the array.

\[
(E_1, \rho) \rightsquigarrow (v_1, \rho') \quad \text{isAry} v_1 \quad (E_2, \rho') \rightsquigarrow (v_2, \rho'') \quad \text{edge}(\rho'', v_1) > v_2 \quad \text{tail}(\rho'', v_1, \#v_2) = v
\]

\[
(E_1 \# E_2, \rho) \rightsquigarrow (v, \rho')
\]

For simplicity, we do not present the cases that errors occurs. Each of them should deal with an abnormal operation. For instance, if the index is out of range, the error is thrown to indicate the illegal access. The rule below describes how to assign a value into a component of a composite expression.

\[
(E_2, \rho) \rightsquigarrow (v_2, \rho') \quad \neg \text{isErr} \rho' \quad (\text{elmAt}(E_1, v_2) \leftarrow v, \rho') \rightsquigarrow (v_1, \rho'')
\]

\[
(E_1 \# E_2, \rho) \rightsquigarrow (v_1, \rho)
\]

\[
(E_2, \rho) \rightsquigarrow (v', \rho') \quad \text{isErr} \rho'
\]

\[
(E_1 \# E_2, \rho) \rightsquigarrow (v', \rho')
\]

If the evaluation of \( E_2 \) does not generate an error, the operation is pass to a certain component of the expression \( E_1 \). In contrast, the error is returned as the value of this expression.
6.4 Object-oriented languages

The previous section studies a set of imperative languages and defines their operational semantics, but provides no support to object-oriented aspects. This section tries to extend the imperative languages $L_A$ to include object-oriented features. We firstly propose a small object-oriented language, then use its expressivity to define some derived expressions, and finally show how to formally represent inheritance.

6.4.1 Representing object, class and method

To clearly show object-oriented features, the following notions are proposed to logically represent composite objects, classes and methods.

$$
\mathcal{O} = \{ \text{obj}(c, y) \mid c \in \mathcal{C}, y \in \mathcal{Y} \} \\
\mathcal{C} = \{ \text{cls}(c, o, y) \mid c \in \mathcal{C}, o \in \mathcal{O}, y \in \mathcal{Y} \} \\
\mathcal{M} = \{ \text{mtd}(v, E_b) \mid v \in \mathcal{V}, E_b \in \mathcal{E} \}
$$

A composite object, rather than a primitive object (e.g., \texttt{true}), is logically composed of two components: its class and store. Any object links to the class where it is created. Its store is represented as an array, and records a set of values belongs to this object. A class consists of its superclass, a prototype object and a local store. The prototype can be considered as the template of class instantiation. To instantiate a class is to copy its prototype. A method is composed of a value in $\mathcal{V}$ and an expression in $\mathcal{E}$. This value represents the result that evaluating arguments generates. The expression stands for method body. The function $\text{getBody} : \mathcal{M} \rightarrow \mathcal{E}$ returns method body for a given method.

To give formal description for method invocation, we define the following auxiliary expression-involution.

$$
\mathcal{M}_k = \{ \text{invoke}(m, v) \mid m \in \mathcal{M}, v \in \mathcal{V} \}
$$

To invoke a method under a certain environment is to assign the value to the temporary store, and
execute its body in the updated environment.

\[
E = \text{getBody } v_1 \quad v_3 = \text{getStack } \rho \quad (E, \text{setStack}(\rho, v_2)) \Rightarrow (v_4, \rho') \\
\text{invoke}(v_1, v_2), \rho \Rightarrow (v_4, \text{setStack}(\rho', v_3))
\]

6.4.2 An object-oriented language \( L_O \)

The object-oriented language \( L_O \) is proposed as follows. \( E_A \) stands for the syntax in \( L_A \).

\[
E = \text{this} \mid l \mid \text{new } E \mid E.E \mid E \# E \mid E_A
\]

- **this**: the current object
- **l**: attribute
- **new E**: create a new object
- **E.E**: attribute selection
- **E \# E**: expression application

The first rule formalizes the reduction for the keyword `this`. Without any premises, its value can be directly obtained from the environment.

\[
\text{(this, } \rho) \Rightarrow \text{getThis } \rho, \rho
\]

The update of an attribute is to change its link to a new value. The value of an attribute can be obtained from the object that the current environment links to.

\[
\text{(l } \leftarrow v, \rho) \Rightarrow \text{null } \text{setVal}(\rho, l, v)
\]

\[
(l, \rho) \Rightarrow \text{getVal}(\rho, l, \rho)
\]

Diagram 6.2 clearly shows how to update an attribute. In that case, the attribute age is relinked from 30 to 31.

For the expression `new E`, the following rules present that if \( E \) reduces to a class, the result of this expression is the copy of the prototype in the class. The global store is extended to include this newly produced object. Otherwise, if a non-class value is generated, an error is thrown to indicate
the unexpected result. Finally, if the evaluation of $E$ generates an error, it will be directly returned as the value of this whole expression.

\[
(E, \rho) \leadsto (v, \rho') \\
isCls v \quad v = cls(c, o, y) \quad \rho'' \cdot o' = clone(\rho', o) \\
(new E, \rho) \leadsto (o', \rho'') \\
(E, \rho) \leadsto (v, \rho') \\
\neg isCls v \quad \neg isErr \rho' \\
r = err "not class" \\
(new E, \rho) \leadsto (r, setErr(\rho', r)) \\
(E, \rho) \leadsto (v, \rho') \\
isErr \rho' \\
(new E, \rho) \leadsto (v, \rho')
\]

The reduction of $E_1 \cdot E_2$ evaluates $E_1$ firstly. If the generated result is not an error, it replaces the current environment. Under this new environment, the expression $E_2$ is evaluated. After evaluation, the original environment is recovered. Otherwise, the error generated is returned as the result of this expression.

\[
(E_1, \rho) \leadsto (v, \rho') \\
\neg isErr \rho' \\
v' = getCntEnv(\rho') \quad (E_2, setCntEnv(\rho', v)) \leadsto (v'', \rho'') \\
(E_1 \cdot E_2, \rho) \leadsto (v'', \rho''') \\
(E_1, \rho) \leadsto (v, \rho') \\
(E_1 \cdot E_2, \rho) \leadsto (v, \rho')
\]

Note that this language does not separately handle class attribute or instance attribute. To properly evaluate class attributes, they can be represented as the form like c.l. With this approach, class attributes can share a common approach with instance ones.

For the expression $E_1 \# E_2$, $E_1$ is evaluated firstly. If the produced value is a method, $E_2$ will be evaluated. Before method invocation, the value of this in the environment is changed. With the values of $E_2$, invoke is used to execute that method. After invocation, this method takes responsibility of removing return error and obtain the returned value from that instruction error. A normal value applied on the function getRtnV returns itself. Finally, the original environment is recovered.

\[
(E_1, \rho) \leadsto (v_1, \rho') \\
isMtd v_1 \quad (E_2, \rho') \leadsto (v_2, \rho'') \\
\neg isErr \rho'' \\
v_4 = getThis \rho'' \\
\quad (invoke(v_1, v_2), setThis(\rho'', getCntEnv(\rho''))) \leadsto (v_3, \rho''') \\
(E_1 \# E_2, \rho) \leadsto (getRtnV v_3, setErr(setThis(\rho''', v_4), null))
\]
The following rules deal with some abnormal cases. The first one presents that the error $E_1$ produces is returned directly. The second one describes if the evaluation of $E_2$ yields an error, it will be regarded as the result of this expression.

\[
\begin{align*}
(E_1, \rho) \rightsquigarrow (v, \rho') & \quad \text{isErr } \rho' \\
(E_1 \neq E_2, \rho) \rightsquigarrow (v, \rho') \\
(E_1, \rho) \rightsquigarrow (v_1, \rho') \quad \text{isMtd } v_1 \quad (E_2, \rho') \rightsquigarrow (v_2, \rho'') \quad \text{isErr } \rho'' \\
(E_1 \neq E_2, \rho) \rightsquigarrow (v_2, \rho'')
\end{align*}
\]

This language allows the expression $E_1 . E_2$ to be assigned. The process can be formalized as follows.

\[
\begin{align*}
(E_1, \rho) \rightsquigarrow (v_1, \rho') & \quad \neg \text{isErr } \rho' \quad v' = \text{getCntEnv } \rho' \quad (E_2 \leftarrow v, \text{setCntEnv}(\rho', v_1)) \rightsquigarrow (v_2, \rho'') \\
(E_1 . E_2 \leftarrow v, \rho) & \rightsquigarrow (v_2, \text{setCntEnv}(\rho'', v')) \\
(E_1, \rho) \rightsquigarrow (v', \rho') & \quad \text{isErr } \rho' \\
(E_1 . E_2 \leftarrow v, \rho) & \rightsquigarrow (v', \rho')
\end{align*}
\]

If $E_1$ does not reduce to an error, the value generated is set into the environment. The update of $E_2$ takes place in the new environment. After the operation, the previous environment is recovered. Otherwise, the generated error is returned as the result of this expression.

### 6.4.3 Derived expressions

So far we propose a small object-oriented language. Although it only provides a number of instructions, this language possesses enough expressivity to represent many derived ones. This section defines some derivation from that small language.

As a pure object-oriented language, FLEXIBO represents arithmetical and logical operations in an object-oriented way. A binary operation can be considered as the first operand receiving a message with the second operand as the argument. For instance, the expression $E_1 + E_2$ is interpreted as the expression $E_1$ receiving the message add with the expression $E_2$ as an argument.

The following list shows a number of binary operations and their corresponding object-oriented representation. All the arithmetic can be defined like the first form. Similarly, logical operations
Unary operations can be considered as special cases of binary ones, but only have single operands. They can be interpreted as a single operand receiving a message with the argument null. For example, the operation-negation is represented as follows.

\[ \neg E = E . \text{not} \# \text{null} \]

The approach has the following advantages. Firstly, it enables the language to have a small and compact kernel, therefore cleanly define its formal semantics. Since these expressions can be represented through the way above, their operational semantics is derived from the fundamental terms. For instance, the semantics of addition can be presented through attribute selection and method invocation. Secondly, it provides the support to the mechanism-operator overloading. Since these operations correspond to a set of fixed-name messages, to overload these operators is to redefine the methods with specified names.

### 6.4.4 Representing inheritance

Object-oriented language regards inheritance as a key mechanism. It provides reusability and extendibility. Through inheritance, a subclass can extend and redefine the attributes in its superclasses. This part formally represents inheritance for this language.

By definition, a class can be represented as the form: \( \text{cls}(c, \text{obj}(c', y'), y) \). Each class records its direct superclass \( c \), a prototype and class store represented as an array. Let \( c_1 \) and \( c_2 \) denote the following classes.

\[ c_1 = \text{cls}(c, \text{obj}(c_1, y'_1), y_1) \]
\[ c_2 = \text{cls}(c, \text{obj}(c_2, y'_2), y_2) \]

\[ E_1 + E_2 = E_1 . \text{add} \# E_2 \]
\[ E_1 \land E_2 = E_1 . \text{and} \# E_2 \]
\[ E_1 : E_2 = E_2 . \text{check} \# E_1 \]
If the class $c_2$ inherits $c_1$, it can be represented in the following form:

$$c_2 = \text{cls}(c_1, \text{obj}(c_2, y_1 \oplus y_2), y_2)$$

According to the form, each derived class only records its direct superclass. This prototype in $c_2$ has a link to the class. Its store is the merge of the one of the object in $c_1$ and the original one in $c_2$. So when the derived class intends to generate an object, it does not need to instantiate its superclass, but directly copies its prototype. Each class possesses a store to locally contain the values of static attributes.

### 6.5 Summary

This chapter formally defines operational semantics for this language. The semantics concentrate on program execution, therefore ignore the features such as variable and attribute declaration. An environment is essential to properly define reduction. It is composed of global store and temporary space. It can be modeled as directed graph. In general, the environment is in charge of a series of store-based and error-based operations.

To avoid defining a complicated semantics, the chapter incrementally shows the formal definition for FLEXIBO. Initially, it proposes a small imperative language and give its operational semantics. Based on this primitive language, an extended imperative language provides supports to loop, error handling and other features. The next language extends the last one through supporting composite data structure-array. Through adding a few instructions, this language possesses object-oriented features. Some expressions can be derived from these fundamental instructions. For instance, it is able to represent arithmetic and logical operations, class inheritance.

In fact, this semantics is only to show the implementation details, does not give too much illustration on language properties. To meet this goal, further study needs to focus on other mathematical semantics. In details, the major challenge of the new semantics comes from how to represent FLEXIBO's pointers. Many research work [44] concentrates on pointer representation. We will evaluate
these results to select an ideal approach for FLEXIBO's modeling. Under the fundamental support, it becomes straightforward to describe FLEXIBO's attributes and addresses. Since this language regards everything as an object, a class directly corresponds to an object with some special properties. After that, we are able to model the relation between objects. For example, the relation clone shows how to make a copy for a given object. Inheritance is also an interesting relation between classes. The semantics will formally reveal how FLEXIBO represents inheritance through its object model, therefore study its mathematical properties.

More importantly, the new semantics enable us to investigate semantic equivalence and full abstraction. The former studies whether the programs with different syntax have identical semantics. In other words, what kind of different programs correspond to a same value in the domain of semantics? Technically, semantic equivalence for operational semantics concerns bisimulation. For algebraic semantics, it is built on the normal form technique. Equal programs are always transformed to the same normal form. Full abstraction is the technique to check whether the mathematical semantics exactly matches its operational semantics. In details, it mainly concerns the mathematical properties: abstractness, soundness, completeness, etc [65]. Their study formally validates the consistency of algebraic semantics and corresponding operational semantics. Practically, it can be used to check the correctness of the operational semantics proposed in the chapter.
Chapter 7

Conclusion and further work

This thesis first identifies the principles for programming languages to support rapid prototype development, and evaluates the suitability of several widely used object-oriented languages. Secondly, we design and implement a dynamically typed language-FLEXIBO based on these principles. It possesses extreme expressivity and flexibility, therefore allows developers to present flexible programs. Thirdly, the thesis proposes an approach for the natural interaction between FLEXIBO and other object-oriented languages such as Java. The most important contribution of the thesis is to design, in a case study, an interpreter that automatically translates specifications coded in FLEXIBO to C programs linked with BSPlib. Using the tool, developers do not need to worry about details of communication, and can concentrate on high-level specification and design. Finally, the thesis formally defines an operational semantics, which describes FLEXIBO’s implementation details.

This chapter also summarizes further work. Firstly, FLEXIBO needs to be systematically tested. This work does not only try to remove language errors, but also make it more suitable for RAD paradigm. Secondly, it is expected to propose a tool to translate FLEXIBO’s programs to C++. This translation can be considered as an approach to fasten FLEXIBO’s performance. Thirdly, we need to design multiple translators for different goals. This work tries to design several translators for different proposes. Fourthly, more support is needed for decentralized programming development. FLEXIBO can control some physical resources like time and memory, but lack the support to “conceptual” resources (e.g., object amount). With more support on these aspects, developers are able
to allocate and control some resource in logic, therefore present high-level resource-related specification. Fifthly, we want to model resource-based calculation through algebraic laws. The formal analysis can be used to validate whether a process overconsumes its allocated resource. Finally, it is expected to define the semantics for programming “compilation”. The previous chapter shows how programs are executed formally, but ignores the issue about the compilation of FLEXIBO's programs. The to-be-defined semantics concentrates on formally describing how to bind attributes and determine variables.

7.1 Contribution

Firstly, this thesis identifies the requirements for programming languages to support prototype development, and examines the suitability for several widely used object-oriented languages. An ideal RAD language need to clearly present language features and minimize program ambiguity. To meet this requirement, the languages are expected to remove confusing operations (eg. pointer) and unify similar operations with common approaches. RAD languages need to support pure object-orientation. Object-oriented programs possess great reusability and extendibility. These features are critical to rapid prototype development. To simplify and fasten programming development, RAD languages need to automatically manage low-level operations. In particular, garbage collectors are essential to deal with spare objects. An ideal RAD language needs to provide a clear and readable syntax for programming readability and understandability. Language interaction is an important mechanism for reusing other languages' properties. Seamless interaction naturally integrates local languages with targets ones. It does not involve in extra instructions, or cause complication.

Evaluating several widely used object-oriented languages with the principles above, we find that most of them are not satisfactory. Smalltalk provides great flexibility and expressivity, but has a very special syntax. It is sharply different with most object-oriented languages. This feature limits developers to quickly learn and master it. C++ is the most unsuitable language for RAD. To keep
compatible with C, C++ is not designed as a pure object-oriented programming language. More seriously, it does not provide a build-in garbage collector, but forces developers to deal with memory allocation and release. The simple language-Java presents a set of advantages for RAD, but has a few easy-to-confused mechanisms (eg. attribute binding). Although the object-based languages: Self and Cecil provide strong support to RAD, they still have some unsatisfactory points. For example, Self reuses most Smalltalk’s syntax, which hinders developers to quickly learn it. Cecil requires developers to involve in language interaction, and presents complicated multiple-inheritance.

Secondly, we design and implement a pure object-oriented language for RAD. FLEXIBO removes most easy-to-confused concepts, and has an unified approach to bind attributes by default. This language pursues pure object-orientation. All the values, even classes and methods can be generally regarded as objects. FLEXIBO hides all the lower-level operations, and is able to automatically collect spare objects. To guarantee programming understandability, FLEXIBO uses a simple and readable syntax. To minimize redundant code, FLEXIBO enables developers to explicitly identify the “color”s for different variables. This language naturally interacts with Java. It allows developers to import Java class, declare Java object, and inherit from the target language.

Moreover, FLEXIBO provides a few mechanisms to make itself more easy-to-use. It uses a flexible exception-handling mechanism to catch both a kind of exceptions and unpredicted exceptions. This language provides an on-line design environment and proposes a Unix-like ownership mechanism to guarantee secure and safe collaboration. To compensate the lack of static type checking, FLEXIBO uses dynamic type constraint to simulate its functionality. It also allows developers to override the default checking rules.

FLEXIBO provides multiple binding ways for attribute determination. Like most dynamically typed languages (eg. Smalltalk), it naturally supports dynamic dispatching. This way enables developers to easily present polymorphism, therefore reduce redundant code. Distinguished with other languages, it also allows binding attributes “statically”. The attributes labelled with at:t are bound to the environments where they are declared. It prevents methods’ semantics from being modified.
because of class inheritance. The def attributes are used to explicitly identify binding environments from attributes. They are particularly useful for using some overridden methods.

Thirdly, this thesis proposes an approach to interact FLEXIBO with other languages. For instance, it presents the features for translating primitive values, cross-language inheritance, etc. We exemplify the technique through FLEXIBO and Java. It does not cause extra instructions for language interaction. To use cross-language properties is almost same as the local ones. More importantly, this process is transparent to developers, does not need their involvement. Technically, it builds two mapping table to translate the data between local languages (eg. FLEXIBO) and target languages (eg. Java). To explicitly determine a method from a group sharing a common name, it tries to extract extra information from arguments. Cross-language inheritance extends reusability from other languages. This mechanism enables local data to have more specified data type in target languages.

Fourthly, the thesis presents a tool to translate a specification to an executable parallel C program. FLEXIBO provides a novel mechanism named reflection. Each reflection system consists of the classes that override FLEXIBO's own classes of program constructs. A common FLEXIBO program can be converted into a kind of reflected programs under different given reflection systems. When the new reflected program is evaluated, user-defined evaluation method instead of the pre-defined method will be invoked. With this mechanism, we design a translator to automatically transfer a specification to its corresponding C code. This process needs four steps. Firstly, the translator verifies the give programmes are well-formed. For a well-formed specification, the translator is able to check communication interference among multiple processes. To produce type-safe target programs, the third step validates type consistence. The final step is to produce the target code from a given specification. Each step of the translation corresponds to a reflection system.

It also defines algebraical laws for each step, and formally prove the correctness of those translations. A few normal forms are used to illustratively verify the translator. In general, it firstly deals with formal translation for primitive instructions with a context-free normal form. Each step of
translation directly corresponds to mathematical computation. Translating more advanced instructions extends that basic normal form with environment involvement. The abstract functions for each step of translation may trigger side-effect.

More importantly, the implementation of this translator states the feasibility to translate a high-level FLEXIBO’s program to a more effective language (e.g., C++). Similarly, we can design a more general translator to “compile” FLEXIBO’s programs to other languages. From FLEXIBO’s perspective, the execution of its programs products a corresponding program of other language. But from the viewpoint of target languages, it can be regarded as a “compilation”. In the other word, FLEXIBO’s programs is “compile” once during the translation, and “run” anywhere in the platform of target languages.

Finally, an operational semantics is formally defined the language. Rather than presenting complex semantics, we incrementally exhibit its formal definition. At the beginning, it launches a small imperative language and its semantics. Based on this primitive language, an extended imperative language provides supports to loop, error handling and other important features. To support composite data structure, another extended language is presented to include the structure-array. With a few extra object-oriented instructions, this language can present object-oriented features. Some undefined operations can be derived from the fundamental instructions.

7.2 Further Work

This experimental language is still under development. More features are subsequently added in. Although we develop a relatively constructive case to check this language, FLEXIBO has not been tested systematically. At least, there exist the following points that need to be checked to guarantee its correctness and robustness. Firstly, we will try our best to remove language bugs. A reasonable approach is to encourage more developers to use FLEXIBO to program. Their feedback is very helpful to the removal. Secondly, the language is expected to be polished to be more suitable to
rapid prototype development. To meet the goal, FLEXIBO may need to be added more features according to developers’ feedback. Thirdly, FLEXIBO is expected to be more easy-to-use and easy-to-learn. To learn a new language may be a hard task to developers. We will try to minimize the “pain” through simplifying its syntax and providing more language flexibility.

The second work is to design a translator to “compile” FLEXIBO’s programs to other efficient languages. FLEXIBO provide extreme flexibility to support rapid prototype development. Like most interpreted languages, FLEXIBO does not have high performance. The previous chapters presents the feasibility to translate a specification based on FLEXIBO to a corresponding executable C program. Similarly, we can design a tool(translator) to “compile” FLEXIBO’s programs to more efficient languages(eg. C++), therefore indirectly improve its efficiency.

Thirdly, we want to defines multiple translation systems. If we think the translating technique more generally, each translation system corresponds to a requirement. In the previous chapters, we define a tool for translating BSP-linked specifications to C programs. This tool is composed of several translation systems, which correspond to syntax checking, communication inference detection, coding generation, etc. Clearly, it is unlikely to define a universal translation system for all these requirements, but multiple translation systems for different proposes. We want to define a number of translation systems for common requirements. For instance, a system may correspond to the translation that attaches importance to execution speed, the other is designed for that translation that regards memory minimization as the most important goal.

The fourth work is to provide more support for decentralized programming development. FLEXIBO provides a few mechanisms for this paradigm. For instance, it uses unix-like ownership control mechanism to guarantee safe on-line collaboration, and provides two build-in classes Timer and Memory to manage “physical” resources. However, FLEXIBO lacks the support to “conceptual” resources such as object amounts. With controlling both “physical” and “conceptual” resources, developers can naturally present high-level resource-based specifications qualitatively and quantitatively. Also, these resources can be formally modeled and computed. Each process is allocated
a quota of resources. Formal technique can be used to compute (or approximate) whether a process over-consumes its allocated resources. The way is able to analyze program safeness through abstract functions, therefore avoids resource over-consumption before program execution.

The fifth work is to define the formal semantics for programming "compilation". Normally, interpreted languages do not clearly separate compilation from programming execution. These processes may occur in the same time. FLEXIBO provides many kinds of attributes or variables (e.g., local variables) for improving language flexibility. In order to accurately present how to determine attributes and variables, we want to formally define programming compilation. This semantics is essential to describe the binding details for determining these attributes or variables.

Finally, we try to strengthen the weakness that exists in both FLEXIBO and the thesis. The most significant weakness in FLEXIBO is graphic user interface (GUI). This simple language does not design its own GUI mechanism, but reuse Java's related properties through language interaction. The approach properly works in a local environment. A challenging problem emerges as the design environment is extended to be online. Programs are designed in local browsers, but executed on remote servers. The evaluation results should be returned to developers through a certain approach. Unlike numerical or boolean values, GUI cannot easily be pass from a server to corresponding clients. So far, FLEXIBO does not satisfactorily solve this problem.

Moreover, this thesis separately illustrates the implementation of the transformer for the fundamental BSP commands and advanced commands, and formally states the correctness. The proof for these parts shares a lot of similarity. Their minor differences only exist in the involvement of an updated environment. The thesis does not deal with these cases in an unified approach, therefore cause many repetitive descriptions. Ideally, the advanced commands are built on, or extended from the fundamental ones rather than redefine many functions for the involvement of an argument.

The thesis claims to successfully give a tool for LOGS transformation, but does not demonstrate this contribution through more practical programs. A case study shown in the appendix exhibits
the feasibility of specification transformation, but could be thought too artificial. The ideal exam­
iples include matrix multiplication, dining-philosopher [18] and so on. These cases would be more
convincing to demonstrate the usability of this tool.

The thesis does not launch an official language specification. This document is expected to
include the following issues. Firstly, it needs to clearly explain all language instructions and ex-
emplify their usage. If FLEXIBO has special syntax or semantic, it should clearly demonstrate the
difference and illustrate the reasons. Secondly, the thesis must give a formal definition(eg. BNF)
for language syntax. It constructs the official guideline to use the language, therefore minimize
syntax-based confusion.
Appendix

A FLEXIBO's Example specification and its generated code

(reflect LOGS
  base (
    var p:Int;
    var w:Int;
    var x: LogsArray[Int,50];
    var y: LogsArray[Int,50];
    var linear : OneDimEvenPartition[3];

    par p from linear.low to linear.high do
      for w from (x:linear # p).low to (x:linear # p).high do
        early after(x#w) = before(x#w) + before(y#w)
    }
  ).eval[];

  // C code with BSPLib is generated for 16 sequential processes.
  // Semantic structure is checked.
  // Types are checked.
  // Communication interference is checked for 18 supersteps in total.
  // There are 3 sequential processes in parallel.
#include "bsp.h"
#include <stdio.h>
#include <malloc.h>

int termin[16];
void idle()
{
    int i,n; for (i=0; i<16; i++)
    bsp_put(i, termin, termin, bsp_pid(), sizeof(int));
    do
    {
        bsp_sync();
        n=0; for (i=0; i<16; i++) n += termin[i];
    } until (n==16);
}

void processor0()
{
    int x[50];
    int y[50];
    //-------- shared variable registration ------
    bsp_pushregister(x, 50*sizeof(int));
    bsp_pushregister(y,50*sizeof(int)); bsp_sync();
    //---------- communication barrier ----------
    for (int w = 0; w <= 16; w++)
    {
        x[w] = (x[w] + y[w]);
        bsp_put(0, x, x, w, sizeof(int)); bsp_sync();
        //----------- communication barrier --------
    }
    bsp_popregister(x);
    bsp_popregister(y);
    //-------- waiting for others ----------
void processor1()
{
    int x[50];
    int y[50];
    //-------- shared variable registration -------
    bsp_pushregister(x, 50*sizeof(int));
    bsp_pushregister(y, 50*sizeof(int));
    bsp_sync();
    //------------- communication barrier -------------
    for (int w = 17; w <= 33; w++)
    {
        x[w] = (x[w] + y[w]);
        bsp_put(1, x, x, w, sizeof(int));
        bsp_sync();
        //------------- communication barrier -------------
    }
    bsp_popregister(x);
    bsp_popregister(y);
    //------------- waiting for others -------------
    idle();
}

void processor2()
{
    int x[50];
    int y[50];
/----- shared variable registration -----
bsp_pushregister(x, 50*sizeof(int));
nbsp_pushregister(y, 50*sizeof(int));
nbsp_sync();
// communication barrier
for (int w = 34; w <= 49; w++)
{
    x[w] = (x[w] + y[w]);
    bsp_put(2, x, x, w, sizeof(int));
    bsp_sync();
    // communication barrier
}
bsp_popregister(x);
bsp_popregister(y);
// waiting for others
idle();

// Main Program

typedef void (*Process)();
Process processes[16];

void init()
{
    int i;
    for (i=0; i<16; i++) termin[i] = 0;
    processes[0] = &processor0;
    processes[1] = &processor1;
    processes[2] = &processor2;
for (i=3; i<16; i++)
    bsp_pushregister(termin, 16*sizeof(int));
}

void main()
{
    init();
    bsp_begin(bsp_nprocs());
    processes(bsp_pid())[i];
    bsp_end();
}
Bibliography


[27] Jin Song Dong and Roger Duke, Using object-z to specify object-oriented programming languages.


[34] Adele Goldberg and David Robson, Smalltalk 80: The language, 1st ed., Addison-Wesley Professional, January 1989.


