

# A Language of Flexible Objects

Yifeng Chen

Department of Computer Science,  
University of Leicester, Leicester LE1 7RH, UK  
Email: Y.Chen@mcs.le.ac.uk

## ABSTRACT

In this paper, we introduce a new language called  $\mathcal{F}$ LEXIBO.  $\mathcal{F}$ LEXIBO is an executable object-oriented specification language designed for open-source software development with different levels of *trust* in a *decentralized* programming environment.  $\mathcal{F}$ LEXIBO provides a number of new programming mechanisms for both systematic static analysis and dynamic control of *correctness*, *ownership* and *resources*. All language ingredients are values. Many programming constructs have their corresponding *mirror classes*, each of which can be extended to programmer-defined sub-mirrorclasses. Various language operators such as method invocation are overridable. Unlike other OO languages that follow specific binding rules,  $\mathcal{F}$ LEXIBO allows programmers to choose the binding mechanism of a variable by explicitly denoting its binding direction. Variables are untyped in  $\mathcal{F}$ LEXIBO. Types are introduced as constraints and checked in runtime. However, if  $\mathcal{F}$ LEXIBO is used for language translation, the checkings are actually done in the compilation phase of the whole process. Programmers are allowed to define their own types. Each type system then corresponds to a particular  $\mathcal{F}$ LEXIBO program. That means  $\mathcal{F}$ LEXIBO programs are able to compile themselves: a source  $\mathcal{F}$ LEXIBO program can be evaluated for type checking and translated to another programming language, if the type checking succeeds. Unlike other languages providing one particular set of language mechanisms,  $\mathcal{F}$ LEXIBO is designed to represent various language mechanisms systematically and flexibly.

## 1. INTRODUCTION

Like many other commercial or non-commercial activities, software development for common-purpose applications will eventually be organized on Internet. Traditional programming languages and environments provide little support for this kind of software development. In this paper, we introduce a new language called  $\mathcal{F}$ LEXIBO.

$\mathcal{F}$ LEXIBO is an executable object-oriented specification language designed for open-source software development with

different levels of *trust* (i.e. the trusting relationships between developers) in a *decentralized* programming environment (i.e. the decentralization of developer groups [22]). Note that, in principle, the produced software is not restricted to any particular paradigm, although our current realization of  $\mathcal{F}$ LEXIBO (see URL [24]) only supports the decentralized development of sequential programs. In the current implementation, both developers and software users are clients of an online server on which  $\mathcal{F}$ LEXIBO programs are interpreted. Various activities (such as coding, testing and execution) of different participants may occur simultaneously in the environment.

In such a dynamic programming environment, guaranteeing software quality becomes a real challenge. Instead of attempting to eliminate all program bugs, we believe that a more plausible solution is to allow bugs to exist, but at the same time, limit their effective scopes using controlling mechanisms.  $\mathcal{F}$ LEXIBO provides a number of new programming mechanisms for both systematic static analysis and dynamic control of *correctness*, *ownership* and *resources*.

In order to support sophisticated controlling schemes, *flexibility*, as the main characteristic of  $\mathcal{F}$ LEXIBO, is pursued throughout the design of the language. For example, the language is object-based (see similar languages and models [1, 2, 4, 6, 14, 11]) in the sense that all language ingredients are *values* (i.e. data manipulatable by code), including primitive values, objects, classes, methods and quoted program code. Many programming constructs have their corresponding *mirror classes*, each of which can be extended to programmer-defined sub-mirrorclasses — a feature called *objects in different shapes*. Various language operators (e.g. method invocation) then become overridable. Unlike other OO languages such as C++, Java and Eiffel that follow specific binding rules,  $\mathcal{F}$ LEXIBO allows programmers to choose the binding mechanism of a variable by explicitly denoting its binding direction — a feature called *variables with different colors*, which has solved some long-standing problems of variable binding. Variables are untyped in  $\mathcal{F}$ LEXIBO. All consistencies are checked in runtime. However, if  $\mathcal{F}$ LEXIBO is used for compiler design and language translation, programmers can introduce their own types — a feature called *types of different kinds*. Such checking is in fact done for the compilation phase of the translation.

Language design is a tradeoff between flexibility and efficiency. In both compiled and interpreted languages, the process of program code consists of two major phases: the

**Figure 1: Basic patterns of program process**

*once phase* in which the program code is analyzed syntactically with some of its types checked statically, and the *all phase* in which the program is executed in runtime with its constraints tested dynamically<sup>1</sup>. In general, it is always desirable to conduct more analysis in the *once phase*, since for any round of the *once phase* (completed by the developers), the users may repeat the *all phase* many times, and in each repetition, a loop body may be iterated millions of times. C.A.R. Hoare has been encouraging researchers to study *the grand challenge of verifying compilers* [17]. The essence of the challenge is to integrate various techniques and tools (including theorem provers and model checkers) to deal with the *once phase* so that program code can be checked as much as possible before the *all phase* (i.e. “once for all”).

Flexibility is certainly desirable, but if a language is too flexible, few consistency checkings and optimizations can be done before runtime. This has limited the extent of existing object-based languages’ application in software engineering. Figure 1 (a) shows the traditional pattern of program process: source code in higher-level languages such as C++ and Java is (partially) translated to one another and compiled to machine code and bytecode, which then become executable and interpretable by JVM [15], respectively. However, there does not exist “the best solution” to the “once for all” problem. In order to pursue both flexibility and efficiency, Java follows some sophisticated typing rules that are sometimes counterintuitive: for example, if a method of a class accesses a field (i.e. data attribute) that is declared in the class and overridden by subclasses, the method will reach to the same class instead of the class that the runtime object belongs to (i.e. static binding); but if the method invokes another (overridden “dynamic”) Java method, it will reach to the class of the runtime object (i.e. dynamic binding). Such a mixed design is confusing<sup>2</sup>. On the other hand, Eiffel uses dynamic binding for both kinds of attributes, while C++ uses static binding for both (without considering virtual methods). Table 1 illustrates the subtle differences of C++, Java and Eiffel. The serious implication is that an essentially same program (with slightly different syntax of different languages) may have different outputs in execution!

Binding	C++	Java	Eiffel
data attributes	static	static	dynamic
methods	static	dynamic	dynamic

**Table 1: Different binding mechanisms**

In the traditional pattern of program process, each language provides a particular solution to the “once for all” problem. *FLEXIBO*, however, reflects a radically different view on compilation and execution and offers a more systematic solution.

<sup>1</sup>The two phases are termed, in the literature, as “compilation” and “execution”, which will have new meanings in this paper.

<sup>2</sup>Most Java programmers that the author knows are unaware of this subtle typing rule for variable binding.

On one hand, *FLEXIBO* programs can be interpreted and tested online. Thus *FLEXIBO* is an interpreted programming language. *FLEXIBO* programs (i.e. quoted expressions) are values that accept the evaluation method *eval*, which can be overridden by programmer-defined evaluation methods. Types are values that accept the type-checking method *check*. Although variables are untyped in *FLEXIBO*, type restrictions can be inserted as *constraints* in a program and checked in runtime by invoking the corresponding *check* method.

On the other hand, the features of *FLEXIBO* are so flexible that they allow *FLEXIBO* programs to “compile themselves” and do what compilers do. For example, a type system can be implemented as a *FLEXIBO* program in which programmer-defined methods of type calculation and checking are invoked for the reflected program code as values. If the *FLEXIBO* program of type checking always terminates, then the corresponding type system must have been decidable. The source code of a given *FLEXIBO* program passes the typing rules of another language, if and only if its “execution” under the type-checking interpretation outputs the boolean *true*. Although types are checked in runtime by *FLEXIBO*, if *FLEXIBO* is used for language translation, the runtime type checking becomes the *once phase* of the whole program process. The final translation from *FLEXIBO* program to another language after type checking can also be realized using the reflection of *FLEXIBO*. Compilation and execution become interchangeable! *FLEXIBO* demonstrates an effort towards the higher-level unification of various OO languages. This new pattern of program process is illustrated in Figure 1 (b).

It is desirable for a source *FLEXIBO* program to have the same structure as the target programs so that they are readable and can be compiled together with other developers’ programs in target languages. Another important consistency requirement is to ensure that the direct interpretation of a *FLEXIBO* program outputs the same results as the target programs, if the source code can pass the typing rules of the target languages. This is more challenging to achieve than it appears (see Section 3) and requires *FLEXIBO* to be “richer” (e.g. in binding mechanism) than the target languages. That also means some of *FLEXIBO*’s language mechanisms cannot be *directly* translated to the target languages and requires *additional code structure or library links* to implement the missing mechanisms. Finally, for the sake of downward compatibility, *FLEXIBO* also allows developers to program entirely in the traditional style without using the new language mechanisms.

Section 2 briefly introduces the syntax and the commands of *FLEXIBO*; Section 3 presents a new variable binding mechanism; Section 4 discusses how to create new language mechanisms and override pre-defined language features in *FLEXIBO* programs; Section 5 studies types in *FLEXIBO*.

## 2. THE LANGUAGE

Commands	Evaluation
$e_1 . e_2$	evaluating $e_2$ in the current env $e_1$
$e_1 ! e_2$	invoking $e_1$ with the argument $e_2$
$e_1 : e_2$	typing $e_1$ with the value of $e_2$
$[e_1, \dots, e_n]$	collecting values of $e_i$ in an array
$\{e\}$	block with body $e$
$e_1 ; e_2$	evaluating $e_2$ after evaluating $e_1$
$e_1 := e_2$	assigning the value of $e_2$ to $e_1$
<b>this</b>	the current environment
<b>super</b>	superclass of the current environment
<b>var</b> $e$	declaring $e$ as a variable
<b>quote</b> ( $e$ )	reflecting expression $e$ as a value
<b>while</b> $e_1$ $e_2$	evaluating $e_2$ repeatedly if $e_1$ is true
<b>if</b> $e_1$ <b>then</b> $e_2$	evaluating $e_2$ if $e_1$ 's value is <b>true</b> , or
<b>else</b> $e_3$	evaluating $e_3$ if it is not.

**Table 2: Basic FLEXIBO commands**

The focus of this paper is on the introduction of new language mechanisms not the language FLEXIBO itself. Instead of providing the whole syntax and formal semantics, we shall only discuss about the program constructs, their meanings and applications.

In general, everything in FLEXIBO is an expression, which can be evaluated to a value. Values are constant expressions. Other expressions include *variables* in Table 4, *basic commands* in Table 2, *derived commands* in Table 3 and *advanced commands* in Table 6, where each  $e_i$  denotes an *arbitrary* expressions,  $n$  denotes a *name*, and  $s$  denotes a *string*.

FLEXIBO is designed from the perspective of non-operational (denotational or algebraic [16, 7]) semantics (cf. [9, 8, 10]) and has few syntactical restrictions compared to other popular OO languages and models [1, 23]. Most consistencies (including types) are checked in **runtime**. However, as a FLEXIBO program may compile itself, many runtime checkings are in fact done in **compilation time** with respect to the whole process of language translation.

In the combinator  $e_1 . e_2$ , the evaluation of  $e_1$  is before the evaluation of  $e_2$ . The value of  $e_1$  becomes the *current environment* for  $e_2$ 's evaluation. The evaluation of  $e_1 ! e_2$ , however, is the invocation of the value of  $e_1$  (evaluated first) with the argument  $e_2$  (evaluated afterwards). Expressions on both sides of “.” and “!” are **arbitrary** expressions. For example, the expression  $o . (m![x, y])$  corresponds to method call  $o . m(x, y)$  in other languages. The difference is, in FLEXIBO, the order of side effect is  $o \rightarrow m \rightarrow x \rightarrow y$  not  $x \rightarrow y \rightarrow o \rightarrow m$  like Java. We believe that FLEXIBO's evaluation order is the correct one, because it can be unified with the order after overriding the method of invocation. Runtime exceptions are generated if the value of  $o$  is not an environment whose attributes are modifiable, the attribute  $m$  does not exist in  $o$ , the value of  $m$  is not something invocable, the invocation fails, or any other errors occur. Variable declaration **var**  $e$  allow standard accessibility keywords such as **static**, **dynamic**, **private**, **protected** and **public**.

Some commands can be derived from the basic ones. For ex-

Commands	Evaluation
<b>clone</b> $e$	creating a clone of the value of $e$
<b>print</b> $e$	printing the value of $e$
<b>eval</b> $e$	evaluating the quoted expression $e$
<b>return</b> $e$	return the value of $e$
<b>new</b> $e$	new object of class $e$
<b>class</b> $e_1$ $e_2$	new class inheriting $e_1$ with body $e_2$
<b>method</b> $e_1$ $e_2$	new method with arg $e_1$ and body $e_2$

**Table 3: Derived FLEXIBO commands**

ample, **clone**, **print**, **eval**, **return** and **new** are overridable predefined methods, and **class** and **method** are shorthand expressions for creating new objects of the mirror classes *Class* and *Method* respectively. For example, the expression **print**  $e$  is the same as  $e . print []$ , and the expression (**method**  $x$   $x$ ) is the same as

**new** *Method* [**quote** ( $x$ ), **quote** ( $x$ )].

The following program illustrates the typical uses of the commands:

```

var  $c :=$  class Value (
    var dynamic  $x : Int$ ;
    var  $m :=$  method [ $y$ ] return  $x + y$  );
var  $o :=$  new  $c$ ;
 $o . x := 0$ ;
print  $o . x$ ;
print  $o . m[1]$ ;
print (new  $c$ ) . ( $1 + m[2]$ );
{ var  $o := 10$ ;
while ( $o > 0$ )  $o := o - 1$ ;
if ( $o == 10$ ) then print  $o$  else print “end” };
while true { };

```

In the above program, the variable  $c$  is assigned to a class extending the root class *Value* (i.e. the mirror class of all values). The variable  $x$  is declared to be dynamic and typed as an integer, which is checked at every assignment attempt in runtime. By default, attributes declared in a class are static, so is the method  $m$ . The (untyped) variable  $o$  is initialized to a new object of class  $c$ , its attribute  $x$  is set to be 0, and the method  $m$  is called on the new object with an argument 2. Note that  $m[1]$  is a shorthand of  $m![1]$ . In the following block, another variable with the same name  $o$  is declared as a local variable and reduced to 0 by the following terminating **while** loop. The next conditional command has the standard meaning. Finally, a non-terminating loop follows the block. If this program is executed in the online environment, it will terminate and generate a timeout exception in the end, because the time of a client program, as a resource, is controlled by the server-side FLEXIBO program against potential denial-of-service attacks from untrusted code. Similar controls apply to the relationship between the implementor and the contractor of a specification.

### 3. VARIABLES WITH DIFFERENT COLORS

An important issue in the new pattern of program process is

to ensure that the interpretation of a  $\mathcal{F}$ LEXIBO program before translation should be the same as the execution of the target code, if the  $\mathcal{F}$ LEXIBO source code passes the type checking of the target language. This is more difficult to achieve than it appears. As we explained before, the same OO program may be interpreted differently in different languages due to different binding rules. When a programmer uses a variable in a program, they are clear about their own intention and know which variable declaration to bind.

Both static binding and dynamic binding are useful in real applications. Dynamic binding is a powerful and flexible concept that allows new software components to reuse the existing implementations and override them when it is needed. However, that also means the meaning of an existing method may not be fixed on declaration and subject to future changes for subclass objects. A developer may not fully aware of the invocation relationships between the methods of the parent class. Overriding a method may unintentionally modify the expected behavior of the parent class. Dynamic binding also tends to cause more overhead in execution than static binding. Eiffel has a compilation option of *finalization* to tell the compiler to transform dynamic binding of non-overridden attributes into static binding. This option works well if all classes are always compiled together but is not applicable for pre-compiled class libraries and component-based development. On the other hand, static binding has a fixed semantics and tends to be more efficient in execution.

In existing OO languages, if attributes with the same variable name appear in both the current class and its subclasses, it becomes impossible for compilers to figure out the exact intention of the programmer. Each OO language makes a particular assumption (see Table 1). For example, most Java programmers are unaware of the subtle difference between static binding of fields and dynamic binding of methods in Java. Inconsistency between compiler’s assumption and programmer’s intention becomes a potential source of software faults. Even if programmers understand the language’s assumptions, sticking to one particular binding rule may cause difficulties where the opposite binding rule is needed. This suggests the lack of expressiveness of the conventional binding mechanisms.

Unlike previous languages that provide one particular solution,  $\mathcal{F}$ LEXIBO provides several binding directions for programmers to choose. This new language mechanism is called *variables with different colors*. In  $\mathcal{F}$ LEXIBO, a variable may appear as a *colored* variable **att**  $x$ , **def**  $x$ , **loc**  $x$ , **arg**  $x$  or **env**  $x$  where the keyword before the variable indicates the binding direction, or appear as a *colorless* variable  $x$  where the default binding direction applies. Note that, unlike C++’s mixed binding mechanism (with virtual methods), the choice of binding directions in  $\mathcal{F}$ LEXIBO is not made upon variable declaration but at variable access. A declared attribute may be accessed from different binding directions.

### 3.1 Variable color for vertical binding

By “vertical binding”, we mean that a variable is bound to the attribute with the same name declared in the class that is *closest* to the *current object*’s class in the hierarchy of inheritance.

Variables	Binding mechanisms
<b>att</b> $n$	variable $n$ of vertical binding
<b>def</b> $n$	variable $n$ of horizontal binding
<b>loc</b> $n$	local variable $n$ of horizontal binding
<b>arg</b> $n$	argument variable $n$ of horizontal binding
<b>env</b> $n$	env variable $n$ of horizontal binding
$n$	variable $n$ of context-dependent binding

Table 4: Variables with different colors

The following program is a real example showing the need for vertical binding (of data attributes in particular).

```

var Person := class Value (
  var message := “I speak natural language.”;
  var speak := method [] print att message;
  var set := method [s] att message := s;
)
var Chinese := class Person (
  var message := “I speak Chinese.”;
)
var HongKonger := class Chinese (
  var message := “I speak Cantonese.”;
)
var British := class Person (
  var message := “I speak English.”;
)
var p : Person;
p := new HongKonger;
p. att speak [];

```

In the above example, the data attribute *message* is overridden in every subclass. The method *speak* in the class *Person* prints the value of the variable **att** *message* through vertical binding. The method *set* can be used to update the message in the current object’s class. The local variable *p* is declared as a person and assigned to a new hongkonger. Since the runtime type of the variable is *HongKonger*, the printed result is “I speak Cantonese.” as expected.

Java uses static binding for fields (i.e. data attributes). The above program does not correspond to any Java program with the same structure. Using **this** does not solve the problem, because **this**.*message* is also statically bound (and it should be so for conceptual simplicity). Java methods are dynamically bound, but using a method to return the message as a constant would prevent other methods such as *set* from updating the message. Java’s type casting cannot solve the problem either, as the cast type must be a constant type. The only plausible solution that we know is to copy the method *speak* in every subclass so that it can access the current object’s message. This, however, renders the code difficult to reuse and maintain. Note that the same phenomenon still arises if the attribute *message* is dynamic.

### 3.2 Variable colors for horizontal binding

By “horizontal binding”, we mean that a variable is bound to the attribute with the same name declared in the environment that is *closest* to variable in the program text. Here, an “environment” can be a block with local variables, a method with arguments or a class with attributes.

The following program is a real example showing the need for horizontal binding (of methods in particular).

```

var SimpleAccount := class Value (
  var dynamic current := 100;
  var amount :=
    method [] return def current;
  var payable :=
    method [a] return (def amount [] >= a);
  var indebt :=
    method [] return (att amount [] < 0);
);
var Account := class SimpleAccount (
  var dynamic savings := 200;
  var amount :=
    method [] return (def current + def savings);
);
print (new SimpleAccount). att payable [200];
print (new Account). att payable [200];
print (new Account). att indebt [];

```

In the above program, the variable *SimpleAccount* is declared to be a class with an attribute *current* recording the balance of the current account. The method *amount* retrieves the balance. The method *payable* tests whether a bill of amount *a* can be paid. The method *indebt* checks whether the total amount of the account is in debit. The class *Account*, however, consists of one more attribute for savings. The total amount is the sum of the current balance and the savings. For a simple account, a bill of amount 200 is obviously not payable; for a general account with savings, such a bill should still not be payable, because in the real world, a bank normally only considers the balance of the current account in direct payment. This is consistent with the above program in which the method *payable* invokes the *amount* method declared in *SimpleAccount* through horizontal binding. Even if the current object belongs to the class *Account*, it still compares the current balance and the given amount to be paid. The method *indebt*, however, invokes *amount* through vertical binding to determine whether the *total amount* is below zero.

Java uses dynamic binding for “dynamic” methods<sup>3</sup>. That means the above FLEXIBO program does not correspond to any Java program with the same structure. Casting the type of **this**

((SimpleAccount)**this**). amount()

cannot solve the problem, because type casting is treated as a static type conversion, and the the binding mechanism remains dynamic in runtime. Using **super** does not solve the problem either, as the current object’s class cannot not be fixed in runtime. A plausible solution is to copy the body of the method *amount* in the method *payable*, but this renders the code difficult to reuse and maintain. Note that sometimes both binding directions lead to the same declared attribute (e.g. *current* in the above program), if the attribute is not overridden. In that case, the semantics of **att** *current* is still different from **def** *current*, as the former is open and subject to overriding from other subclasses in

<sup>3</sup>This quoted word “dynamic” is Java’s term regarding the methods that may access dynamic fields. A dynamic attribute in FLEXIBO must be physically stored in objects not their class.

future. Programmers decide either to close the semantics or to keep it open by explicitly denoting the color of a variable.

C++ allows both static binding for common methods and dynamic binding for virtual methods, but the above example does not correspond to any C++ program with the same structure either. In C++, whether or not a method is virtual is determined on declaration. In FLEXIBO, however, the choice of binding direction is determined at variable access. For example, the method of *amount* is accessed through horizontal binding in the method *payable* and vertical binding in the method *indebt*.

If there are several nested environments that declare the same variable, a variable access is always bound to the variable declared in the *closest* environment. The following program shows an example of nested environments.

```

var Vehicle := class Value (
  var brand : String;
  var show := method [] print def brand;
  var Engine := class Value (
    var brand : String;
    var show := method [] print def brand;
    var Valve := class Value (
      var show :=
        method [] print def brand;
    )
  )
)

```

In the above program, the class *Vehicle* has a brand and an inner class *Engine*, which also has a brand and an inner class *Valve*. We assume that the engine and the valve share the same brand. Thus the method *show* in the class *Chip* reaches to the *brand* declared in *Engine* not the one declared in *Vehicle*.

An environment can be a block, a method or a class. The variable **color** **def** “searches” every layer of environment and is similar to the static binding mechanism of SCHEME [3]. Note that such “searching” can always be done in the *once phase*, even for FLEXIBO programs, as it can be determined upon semantic analysis.

Besides variable **color** **def**, we also introduce another three colors **loc**, **arg** and **env** of horizontal binding, and they are bound to the closest local variable in blocks, the closest argument in methods, and the closest attribute in classes, respectively. The following program illustrates the applications of these variable colors.

```

var c1 := class Value (
  var x : Int;
  var m1 := method [x : Int] {
    var x : Int;
    var c2 := class Value (
      var y : Int;
      var m1 := method [y : Int] {
        var y : Int;
        print def x + def y;
        print loc x + loc y;
        print arg x + arg y;
        print env x + env y;
      }
    )
  }
)

```

```

)
}
)

```

In the above program, **def**  $x$ , **loc**  $x$ , **arg**  $x$  and **env**  $x$  are bound to the local variable  $x$  in  $m1$ 's block, the argument  $x$  of  $m1$  and the attribute  $x$  in  $c1$ . **def**  $y$ , **loc**  $y$ , **arg**  $y$  and **env**  $y$  are bound to the local variable  $x$  in  $m2$ , the local variable  $x$  in  $m2$ , the argument  $y$  of  $m2$  and the attribute  $y$  in  $c2$ .

### 3.3 Colorless variable and the default rule

FLEXIBO is “downwards compatible” in the sense that programmers can write code in the traditional style without using the new features of the language. In particular, if the color of a variable is not denoted explicitly, a default rule is applied to determine its color according to the context.

For example, in the following anonymous method

```
method [] return  $e_1 . (e_2 . (e_3 ! e_4))$ ,
```

the colorless variables in  $e_1$  and  $e_4$  are horizontally bound (by default) and evaluated in the environment of the method's block, while those in  $e_2$  and  $e_3$  are vertically bound (by default) and evaluated in the environments provided by  $e_1$  and  $e_2$  respectively.

The default binding direction, applying to both data attributes and methods, switches between horizontal binding and vertical binding according to the following rules:

1. horizontal binding on the first entry of a method or a class;
2. vertical binding for the right-hand expression of a dot expression;
3. horizontal binding for the right-hand expression of an expression of invocation.

For example, the method **method** [] { **return**  $a . (b . (c ! d))$  }, is the same as

```
method [] { return def  $a . (att$   $b . (att$   $c ! def$   $d))$  }.
```

The variable colors **loc**, **arg** and **env** are only useful when the program needs to jump to an outer layer and skip another declared variable with the same name. The informal semantics are crystal clear.

## 4. OBJECTS IN DIFFERENT SHAPES

A programming language normally provides a particular set of language mechanisms. In the new pattern of program process, a FLEXIBO source program may be translated to different target programs in different programming languages. That demands FLEXIBO to be “richer” in language mechanisms. This is achieved by allowing programmers to create their own programming mechanisms and override pre-defined features.

Commands	Evaluation
<b>spec</b> $e$	a specification with pre/post conditions
<b>own</b> $e$	protecting $e$ 's value with ownership
<b>error</b> $s$	retrieving the exception with message $s$
<b>try</b> $e_1$	evaluating $e_2$ if an exception with prefix-
<b>catch</b> $s$ $e_2$	string $s$ occurs in the evaluation of $e_1$

**Table 5: Commands of correctness, onwership and resources control and exception handling**

### 4.1 Control of correctness

VDM-like pre/post specifications [18] are an important means of quality control in software engineering [19]. Eiffel is the first language to embrace the use of specifications by providing them as a language mechanism. The following program shows how different “shapes” of specifications can be incorporated in FLEXIBO.

```

Specification := class Method (
  var dynamic  $pre$ ;
  var dynamic  $post$ ;
  var dynamic  $before$ ;
  var  $init := method$  [ $a, pr, po$ ] {
    [ $args, pre, post$ ] := [ $a, pr, po$ ];
     $implementation := quote$ 
      (error “Incomplete implementation.”);
  };
  var  $implement := method$  [ $imp$ ]
     $implementation := imp$ ;
  var  $invoke := method$   $v$  {
     $before := clone$  this;
    var  $result := super . invoke ! v$ ;
    if (eval  $post$ )
      then return  $result$ ;
    else return error “Postcondition failure.”;
  }
);
var  $c := class$  Value (
  var dynamic  $x := 1$ ;
  var  $myspec := spec$  [] ( $x > 0$ ) ( $before . x < x$ );
);
 $c . myspec . implement ! quote$  ( $x := x + 1$ );
var  $o := new$   $c$ ;
 $o . myspec$  [];
print  $o . x$ ;

```

The above program defines a kind of pre/post specifications. The class *Specification* extends the mirror class *Method* (i.e. the class of all FLEXIBO methods) with three data attributes: *pre* for the precondition, *post* for the postcondition and *before* for the clone of the current object before the operation. The method *init* overrides the pre-defined initialization and sets the implementation to generate an exception. A specification can be implemented by setting the implementation attribute to a given expression. The method *invoke* overrides the operator “!”. On any invocation, a specification will first make a copy for the current object, then evaluate the specification as a method, and finally check the postcondition. Cloning is needed because of the implementation's potential side effect to the current object. The pre-defined method *clone* can also be overridden to adopt either shallow or deep cloning. The class *c* consists

of a data attribute  $x$  and a newly defined specification with a precondition requiring the value of  $x$  to be positive and a postcondition requiring  $x$  to increase. The specification is then implemented by an expression that increases  $x$  by 1. The implemented specification is invoked on the new object  $o$  of the class  $c$ . The result of the implementation is checked against the postcondition successfully.

Note that the above program is only a demonstration to show how a simple kind of specification can be defined. The actual design of specifications in  $\mathcal{FLEXIBO}$  needs to be more sophisticated because of ownership and resources controls.

Whether an implementation needs to check the precondition and the caller (i.e. the contractor) needs to check the postcondition depends on the level of trust between the two sides. On one hand, checking serves as a means of testing and additional guard; on the other hand, not every predicative conditions can be expressed as a boolean expression in programs, and there is overhead for doing so. In Eiffel, the control of the checking is an option of the compiler.

In general, the checking of the postcondition should be done in the *all phase*, but this does not mean all the checking must be so. This is where the tools of theorem proving can be integrated into  $\mathcal{FLEXIBO}$ . Indeed,  $\mathcal{FLEXIBO}$  has provided a standard environment in which various tools of static analysis can be developed or integrated.

In a decentralized programming environment with different levels of trust, one specification may enjoy multiple implementations from both trusted and untrusted implementors. The contractor is free to choose the implementation for each invocation. Various scheduling and management algorithms can be applied. Each new shape will correspond to a new subclass of *Method*

The advantage of  $\mathcal{FLEXIBO}$  is that, although the language itself does not provide any mechanisms for correctness control, it allows programmers to define their own kinds of mechanisms for such purposes by extending the mirror class *Method*. Similar programmer-defined mechanisms can be introduced for UML and aspects.

## 4.2 Ownership and resources control and exception handling

Contract-based pre/post specifications have not been widely used in OO programming [21]. However, in a decentralized development environment with different level of trust, such contracts become a necessary and reliable means of communication between developers.

If a contractor cannot fully trust the implementor, the control of correctness cannot entirely prevent the implementation of a specification from doing malicious things to the product software. The contractor must be able to control the access of the implementation to the contractor's objects and variables and the resources consumed by the implementation.

The following program is an example tested in the online  $\mathcal{FLEXIBO}$  environment [24].

```
try Timer := 0;
catch "" print "I can catch the ownership violation.";
var timer := new Timer;
try { timer.start[2000]; while true {} };
catch "" print "I can catch this timeout exception.";
try { timer.start[5000]; while true {} };
catch "" print "I cannot the server's exception.";
```

The above program first tries to modify the value of the variable *Timer*, which is actually declared in the package by the server-side  $\mathcal{FLEXIBO}$  program and owned by the server. An expression (e.g. a variable) following the keyword **Own** is owned by the owner who conducts the semantic analysis. For example, "**var Own** x" declares an ownership-protected variable whose accessibilities to other developers can be set by its owner. Any attempts of unauthorized access generate runtime exceptions. It is also possible to protect a value with ownership. The command "**own** e" protects the value of the expression  $e$  with the current ownership, and accessibility permissions can be set. *Timer* is a class for time controls. The expression "*timer.start*[2000]" starts a timer for 2000ms. The following empty loop generates a timeout exception, which is caught by the **catch** statement. However, starting a timer for another 5000ms before a nonterminating loop will generate an exception that is not catchable by the client. That is because the server side  $\mathcal{FLEXIBO}$  program has already set a timer for 4000ms and plant an ownership-protected exception (i.e. **own error** "Time out") into the timer. The client cannot catch an exception owned by the server (without the server resetting the permissions).

The relationship between the server-side and the client-side  $\mathcal{FLEXIBO}$  programs is very similar to the relationship between the contractor and the implementor of a specification. A contractor can use the same mechanisms to control the implementor's accessibility and resources. Note that the overhead of additional checking only occurs if the values and the variables are explicitly protected with ownership.

Apart from time and memory constraints, many things can be regarded as resources such as the number of files opened, the size of data transmitted in communication and so on. A typical way of setting a specification is to use a method with a specification "hole" and set the ownership and resources restrictions before the invocation of the specification. The method then becomes a "general specification" incorporating restrictions of correctness, ownership and resources.

Note that a contractor may be the implementor of another developer. The contractor-implementor relationships form a directed graph in which the original contract sits at the top, and its constraints are inherited by all components. For example, a stand-alone Java program has unlimited access to the local files, but a Java applet has no access at all. It is more desirable to allow a Java applet to access a limited number of files of limited sized. Unfortunately, traditional languages do not support such kind of flexible resource control well, because that means every piece of code must voluntarily check the constraints when they use resources, and this is impossible in a large software product. In  $\mathcal{FLEXIBO}$ , however, constraints are automatically enforced by the language.

If there are enough specification constraints covering vari-

ous parts of a software product, it is then possible to localize potential faults in the code. For example, if the original contract has time constraint, no non-terminating loop can make the whole application stuck. Software development is then a process to reduce the number of specification holes and the frequency of fault occurrences in the implementations of the specifications. Any intermediate product can be tested in execution. Redundancy with multiple implementations (of a single specification) may also help improve software robustness.

There have been a lot of interests in resources control [5, 13] recently. The focus is on the means to maximize in the *once phase*, mainly using the design of type systems.  $\mathcal{F}$ LEXIBO does not solve the challenges, but it provides the a standard platform on which different solutions can be represented, implemented and tested systematically in an object-oriented style.

Finally, it also becomes possible to integrate modeling languages and the related tools into  $\mathcal{F}$ LEXIBO. For example, a sequence diagram in Unified Modeling Language (UML) may involve multiple objects and methods. It can be incorporated directly as a new (derived) command that is executable in the online environment. Since existing programming languages do not support sequence diagrams directly, code generation algorithms [12] are still needed for the translation from sequence diagrams to the target languages. Such a transformation algorithm can be developed in an object-oriented style.

## 5. TYPES OF DIFFERENT KINDS

Static typing is the main method of checking in the *once phase* of the traditional program process. Type checking helps guarantee the partial safety of program code by catching the majority of common mistakes made by programmers. If the types of expressions can be determined (or partially determined) statically, that can provide useful information to the compiler for optimization purposes. That is, if a checking or an action can be done in the *once phase*, then there is no need to repeat it in the *all phase*. For example, if it can be inferred that an object always belongs to a certain class, then the entry of the method call to that object can be statically determined by the compiler. That saves the overhead of referencing. Type information also serves as an abstract form of documentation for the signature aspect of a program.

An ideal type system should be clear in meaning, flexible for programming and efficient on bugs prevention and compiler optimization. The first goal is of the most importance, as it affects software quality directly. The other two are desirable as well. However, these goals often conflict with each other. For example, the mixed design of Java’s binding mechanisms has certainly addressed several important issues such simplifying the setting and retrieving of fields and flexible invocation of methods, but it is not clear enough to meet the first criterion. Flexibility and ability to eliminate bugs are also common conflicting goals. Obviously not all bugs can be eliminated in the *once phase*. It may be theoretically possible to capture all necessary checkings of the *once phase* using types, but again if the type system is too complicated, programmers will be confused. A good design

of type system is hence a tradeoff among the concerns.

In the traditional style of program processes, one programming language offers only one solution to the checking of the *once phase*. Typing rules are regards as inference rules (similar to inference rules in logic), and type inference is mainly treated as syntactical manipulations.

From our perspective, typing is essentially a computation in the *once phase*. As a higher-level unification,  $\mathcal{F}$ LEXIBO provides a more systematic solution in which each particular type system (of a language) is represented as a  $\mathcal{F}$ LEXIBO program whose evaluation method *eval* is overridden for type inference and checking. It suffices to show the decidability of a type system by proving that the  $\mathcal{F}$ LEXIBO program terminates and outputs either a type error or an inferred type. Similar technique can be used for the translation from  $\mathcal{F}$ LEXIBO programs to a target language: the translation, as a kind of “evaluation”, prints the target code as the output. If a  $\mathcal{F}$ LEXIBO program passes the typing rules of a target language, it is then possible to be translated to the language; otherwise type errors will be generated in runtime, which is actually the compilation time of the whole translation process.

In  $\mathcal{F}$ LEXIBO, variables are untyped, but types can be inserted as constraints and checked in runtime. In general, any type is a value, and any value is a type! That means, an arbitrary expression can be placed where a type is expected. For example, both (var  $x : Int$ ;  $x := 1$ ) and (var  $x : 2$ ;  $x := 1$ ) are perfectly legal  $\mathcal{F}$ LEXIBO programs, although the latter will generate a runtime exception, because it does not provide the necessary “service” of a type. The only service of a provided type is the *check* method, which checks the current type (i.e. the current object) against a given type. Any value, by default, has a *check* method that checks the reference equality between the current type and the given type. In the latter example, the variable  $x$  is declared to have “type” 2, which is not equal to *Int* and hence generates a type error. The following example shows the runtime checking of types.

```
var  $x : String$ ;
 $x :=$  “This is a successful assignment”;
print  $x : (String + String)$ ;
print  $x : Int$ ;
```

In the above program, the assignment is successful, as the runtime type of any string is exactly the same as the value of *String*. The evaluation of the expression  $x : (String + String)$  also succeeds where  $(String + String)$  is equal to *String* and acts as a constraint checked against the type of  $x$ . In the next statement, the expression  $x : Int$  with a type constraint will generate a type error in runtime.

Programmers are able to define their own kinds of types by overriding the *check* method.

```
var MyType := class Value (
  var dynamic type;
  var init := method [ $t$ ] type :=  $t$ ;
  var check := method  $t$  return type >=  $t.getTpe$ [];
);
var Person := class Value (
  var msg := “I am a person.”
```

Commands	Evaluation
<b>reflect</b> $e$	new reflection system
<b>flexibo</b> $e_1 e_2$	evaluating $e_2$ in reflection system $e_1$

Table 6: Reflection system

```

);
var Staff := class Person (
  var msg := "I am a staff."
);
var NewPerson := new MyType [Person];
var s : NewPerson;
s := new Staff;
var p : Person;
p := new Staff;

```

The variable *MyType* is declared to be a new class with a dynamic attribute *type*, a method *init* for attribute initialization and a method *check* for type checking, which returns a boolean *true*, if the given type *t* from the argument is the same as or inherits *type*. Each object of the class *MyType* is a “downwards-closed” version of *type*. For example, let *Person* and *Staff* be two classes with inheritance relationship. We can then declare a new type called *NewPerson*, which stores *Person* as the inner type. The variable *s* is declared as a member of *NewPerson*. Because the *check* method has been overridden, the assignment of a new *Staff* object to the variable succeeds; on the other hand, similar assignment to a variable declared with the type *Person* will generate a type exception.

```

var r := reflect [
  class Assignment (
    var eval := method []
    if (e1 . check! (e2 . eval []))
    then Null
    else error "Type violation."
  ),
  class TypedExp (
    var eval := method [] return t;
  ),
  class PrimitiveValue (
    var eval := method [] return att getType [];
  )
];
print flexibo r (var x : Int; x := x + 1);

```

In the above program, the **reflect** expression creates a reflection system by uploading three classes that extend the reflection classes of assignments, typed expressions (mainly for typed variables) and primitive values (including integers, strings, etc.). The evaluation method in each uploaded class is overridden. The new evaluation methods actually conduct the type inference and checking. For example, the “value” of both the variable *x* (with its type stored in attribute *t* in class *TypedExp*) and the integer 1 is *Int*, their type. The type operation *Int + Int* also returns the type *Int*. The new evaluation method of the assignment does not modify any variable but checks the type of the assigned variable against the *value* of the right-hand expression, which is actually its type. The expression **flexibo** first converts (with an overridable conversion method) the second given expression to a

reflected expression using the reflection system *r*, and then calls the new evaluation method (i.e. the type checking). As no exception is generated in the new evaluation, the given expression has passed the the checking represented by the reflection system *r*. Note that this is only a partial reflection system to demonstrate how *FLEXIBO* can be used as a compiler constructor. If a reflection expression is not extended by any new class, the original evaluation method will be invoked on evaluation. Other methods can also be added in the extending classes. The reflection expression also adjusts the uploaded classes to share static attributes according to the original inheritance relationships between expression classes.

Each reflection system is a *FLEXIBO* program and may introduce a set of new evaluation methods. Different reflection systems represent different type systems. This approach has essentially provided a *systematic* object-oriented way of implementing type inference rules. In type theory, each inference rule has the form  $Assumptions \vdash e : T$  where an expression’s type is deduced under some assumptions (e.g. about the types of subexpressions). For example, the following is a typing rule:

$$e_1 : Int, e_2 : Float \vdash (e_1 + e_2) : Float .$$

The checking of this rule can be implemented as a **if-then** statement that matches the left-hand pattern first and assigns the type to the right-hand expression next. Such **if-then** construct can be naturally distributed to corresponding classes in object-oriented programming and relies on *polymorphism* to do the pattern matching. In *FLEXIBO*,  $e_1$  and  $e_2$  are evaluated for the calculation of their types, the method *add* is then called upon the type of  $e_1$  with the type of  $e_2$  provided as the argument.

Apart from the object-oriented implementation of typing rules, *FLEXIBO* also has the advantage that it now allows programmers to use other programming constructs such as iteration and introduce new attributes (in uploaded classes) for type checking. This is especially useful to programmer-designed new types. It is more straightforward and convenient to use the whole set of programming constructs with richer expressiveness. Developing a type system then becomes the same as writing a *FLEXIBO* program that deals with type values. On the other hand, any *FLEXIBO* program dealing with type values also corresponds to a type system, although it may not be easy to encode such a type system back to inference rules.

Another advantage of the type system’s new object-oriented style of implementation is that the existing *FLEXIBO* features can be reused. For example, in the above example, the evaluation methods of *Add* and other unextended reflection classes remain unchanged from *FLEXIBO*’s own evaluation. If multiple type systems co-exist, it becomes possible to realize their mutual reuses. This can be achieved by overriding the *convert* method in the uploaded classes so that different subexpressions of the given expression may be converted under different reflection systems, and as the result, invocation of one method may delegate sub-inocations to other reflection systems.

Finally, the translation of *FLEXIBO* programs to a target language can be achieved using the same techniques by ei-

ther adding a method *translate* into each uploaded class or introducing an entirely new reflection system.

## 6. CONCLUSIONS AND FUTURE WORK

This paper has presented a number of new programming mechanisms of  $\mathcal{F}$ LEXIBO, although the language's precise syntax and semantics are not discussed in details. It must be emphasized that programmers are allowed to write program code in the old style without using any of the new mechanisms. In this sense,  $\mathcal{F}$ LEXIBO is "downwards compatible". Similar to the .NET strategy [20],  $\mathcal{F}$ LEXIBO is also designed to unify languages, although the former is to unify languages at an intermediate-language level, while  $\mathcal{F}$ LEXIBO reflects an effort towards the unification of higher-level programming mechanisms. In the past, the lack of efficiency has limited the applications of object-based language. The design of  $\mathcal{F}$ LEXIBO shows that if we push flexibility to an extreme, a language can then be used for translator construction, and the efficiency issue can be solved by transforming source code into other more efficient languages. In order to support such translation,  $\mathcal{F}$ LEXIBO's features must be richer than the *cores* of target languages. This is demonstrated in the variables with different "colors", objects in different "shapes" and types of different "kinds". The issue of multiple inheritance is addressed using "virtual attributes" (see [24]). The potential applications of the new programming mechanisms need to be further explored.

$\mathcal{F}$ LEXIBO is designed from the perspective of non-operational semantics (i.e. denotational semantics and algebraic semantics) and has few syntactical restrictions. Syntactical manipulations (e.g. type checking) are represented as  $\mathcal{F}$ LEXIBO programs in an object-oriented style. A predicative semantics [7, 9] will be studied in our future papers. An important objective of semantic studies is to show the safety of the language with respect to the correctness, ownership and resources controls and the monotonicity of software quality in development process.

$\mathcal{F}$ LEXIBO is currently implemented on top of Java. The interface with Java allows Java classes to be used directly in  $\mathcal{F}$ LEXIBO programs. Developers on the client side share the same programming environment provided by the server on which all activities of development, testing and execution happen in a time-slicing manner. No linear search (like Smalltalk [14]) is needed for any binding mechanisms.

The design of  $\mathcal{F}$ LEXIBO is influenced by several well-known languages: C++, Java [15], Eiffel [19], Smalltalk [14] and Scheme [3] (for horizontal binding). Some new mechanisms (e.g. variable colors) are also applicable to other languages (e.g. Java and Eiffel). Future development includes translation from  $\mathcal{F}$ LEXIBO to C++, Remote Method Invocation (RMI) for client-side HCI, and the incorporation of UML diagrams (as  $\mathcal{F}$ LEXIBO commands), their inter-consistency checking and code generation (as translation into other languages).

### Acknowledgment

The author gratefully acknowledges the participation of J. Zhou in the discussions and implementation.

## 7. REFERENCES

- [1] M. Abadi and L. Cardelli. *A theory of objects*. Springer, 1996.
- [2] M. Abadi, K. Rustan, and M. Leino. A logic of object-oriented programs. In *Verification: Theory and Practice*, volume 2772 of *LNCS*, pages 11–41. Springer, 2003.
- [3] H. Abelson, J. Sussman, and G. Sussman. *Structure and Interpretation of Computer Programs*. MIT Press, 2nd edition, 1996.
- [4] N. Adams and J. Rees. Object-oriented programming in Scheme. In *ACM Conference on Lisp and Functional Programming*, pages 277–288. ACM, 1988.
- [5] J. Aldich and C. Chambers. Ownership domains: Separating aliasing policy from mechanism. In *European Conference on Object-Oriented Programming*, 2004.
- [6] A. Borning. Classes versus prototypes in object-oriented languages. In *ACM/IEEE Fall Joint Computer Conference*, pages 36–40, 1986.
- [7] Y. Chen. Generic composition. *Formal Aspects of Computing*, 14(2):108–122, 2002.
- [8] Y. Chen. A fixpoint theory for non-monotonic parallelism. *Theoretical Computer Science*, 308(1-3):367–392, 2003.
- [9] Y. Chen and J.W. Sanders. Logic of global synchrony. *ACM Transactions on Programming Languages and Systems*, 26(2):221–262, 2004.
- [10] Y. Chen and J.W. Sanders. The weakest specifunction. *Acta Informatica*, (to appear).
- [11] W.R. Cook. Interfaces and specifications for the Smalltalk-80 collection classes. In *OOPSLA*, pages 1–15, 1992.
- [12] Rational Software Corporation. *Rational Rose - Using Rational Rose 98*. Rational Inc., 1998.
- [13] W. Dietl and P. Müller. Exceptions in ownership type systems. In E. Poll, editor, *Formal Techniques for Java Programs*, pages 49–54, 2004.
- [14] A. Goldberg and D. Robson. *Smalltalk-80: The Language and Its Implementation*. Addison-Wesley, 1983.
- [15] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification*. Addison-Wesley, 1996.
- [16] C. A. R. Hoare and J. He. *Unifying Theories of Programming*. Prentice Hall, 1998.
- [17] C.A.R. Hoare. The verifying compiler: A grand challenge for computing research. *Journal of the ACM*, 50(1):63–69, 2003.
- [18] C.B. Jones. *Systematic Software Development Using VDM*. Prentice Hall, 1990.
- [19] B. Meyer. *Object-Oriented Software Construction*. Prentice-Hall, 2nd edition, 1997.

- [20] B. Meyer. .net is coming. *IEEE Computer*, 34(8):92–97, 2001.
- [21] B. Meyer. The grand challenge of trusted components. In *International Conference on Software Engineering*, pages 660–667. IEEE Computer Society, 2003.
- [22] P. Oreizy, N. Medvidovic, and R. Taylor. Architecture-based runtime software evolution. In *International Conference on Software Engineering (ICSE'99)*, pages 177–186. IEEE Computer Society, 1998.
- [23] D. Walker. Objects in the pi-calculus. *Information and Computation*, 116(2):253–271, 1995.
- [24] FLEXIBO web site.  
<http://www.bsp-worldwide.org/survey/report.html>.