Inheritance, Specification and Documentation Support for an Object-Oriented Language

By
Jie Liang, B. Sc., M. Eng.

A Thesis
Submitted to the School of Graduate Studies
in partial fulfilment of the requirements for the degree of

M. Sc.

Department of Computing and Software
McMaster University

MASTER OF SCIENCE (2004) McMaster University
(Computing and Software) Hamilton, Ontario

TITLE:
Inheritance, Specification and Documentation Support for an Object-Oriented Language

AUTHOR: Jie Liang, B. Sc., M. Eng. (Northeastern University, PRC)
B. Sc. (McMaster University, Canada)

SUPERVISOR: Dr. Emil Sekerinski

NUMBER OF PAGES: xii, 126

Abstract

Lime is an object-oriented programming language with action-based concurrency. Concurrency in Lime has been previously implemented. Our current work integrates specification and documentation features into Lime and its compiler. The goal of this integration is to improve software quality, in particular correctness, extensibility, and maintainability in a uniform and coherent manner.

Lime provides a flexible inheritance mechanism that separates subclassing (code sharing) from subtyping (substitutability). Any class can sever as a superclass (for the purpose of reuse) or/and as a supertype (for the purpose of specification) and any child class can either inherit the implementation, the interface specification, or both.

Behavioral specifications are expressed by preconditions, postconditions and invariants. These and other intermediate annotations can be written using quantifiers and other standard mathematical notation, and can be checked at run-time.

The associated documentation tool generates a description of the behavioral interface of each class that includes the preconditions and postconditions of the methods, the class invariants, the subtype hierarchy, and subclass hierarchy.

Acknowledgments

First, I would like to express my sincere thanks to my supervisor, Dr. Emil Sekerinski, for his thoughtful guidance and constant encouragement throughout my study.

I am grateful to Dr. Riaha Khedri and Dr. Spencer Smith for their careful review and comments. Thanks to all the other professors and graduate students for their help in these two years.

Thanks to my families for their love, understanding and support.

Contents

.

List of Figures

List of Tables

Chapter 1

Introduction

Lime is an action-based object-oriented concurrent programming language, which was developed by E. Sekerinski from McMaster University [54]. The development of Lime is based on the observation that more and more applications will be implemented on networks of processors in the future. It is difficult for programmers to do multiprogramming using current object-oriented languages. The claim is that action systems, which model concurrency by nondeterministic choices among actions, can help us to simplify both the specification and design of concurrent applications. Lime combines action systems with object-oriented language features to make the development of object-oriented concurrent programs easier and more efficient. Concurrency in Lime has been implemented by a previous year's master student Guanrong Lou [42]. The main goal of our work on Lime is adding some language features to improve software quality.

1.1 Software Quality Factors

The purpose of software engineering is to find ways of building high quality products. Software quality is best described as a combination of several factors. Quality factors can be divided into two different categories, external and internal factors. The definitions in this section are from Meyer's book [45].

External quality factors are such properties as speed or ease of use, whose presence or absence in a software product may be detected by its users.

Internal quality factors are other properties, such as being modular or readable, which are perceptible only to computer professionals who have access to the actual software text.

External factors are best achieved through the internal ones. The intention of our work is to provide a set of language features which hold the potential for improvements in the following quality factors:

- **Correctness** is the ability of software products to perform their exact tasks, as defined by their specification.

- **Reusability** is the ability of software elements to serve for the construction of many different applications.

Other quality factors such as **robustness** and **efficiency** can further be improved in future work by providing exception handling mechanism [14, 21, 38, 49] and code optimization.

An automatic documentation generator may help to meet the requirements of **extendibility, reusability** and **maintainability** by providing information on a class's

application programming interface (API), behavioral specification and the system's structure.

1.2 Important Lime Language Features

In this section, we discuss some of the language features that we have added to Lime to increase its potential for improvement in software quality factors.

The first quality factor we consider to improve is correctness. Correctness is the prime quality. If a system does not do what it is supposed to do, everything else about it, whether it is fast, easy to use etc, means little. Correctness is a relative notion. A software system or its component is neither correct nor incorrect; it is correct or incorrect with respect to a certain specification.

To be correct, a software implementation must have a behavioral interface specification which consists of both the specified interface and the specified behavior.

The interface specification languages in the Larch family [33, 57], JML [10, 12, 34] and Eiffel [45] specify the behavior of their modules by a Hoare-style specification, i.e., preconditions and postconditions [23].

An invariant is a correctness condition imposed on a class, and this condition must not be violated by any method of a class. A precondition is associated with a particular method and imposes a correctness condition on the client of the method; the client must ensure that the precondition is fulfilled. A postcondition is also associated with a particular method, but it imposes a correctness condition on the implementation of the method; a violation of a postcondition indicates an error in the implementation of the method [50].

We added preconditions, postconditions and invariants to the Lime language to

formulate behavior specifications; the main advantage of this approach is the homogenous integration of assertions into the programming language, i.e., compiler error messages are consistent. For the consideration of the tradeoff between correctness and efficiency, we made the Lime compiler have the capability to selectively enable and disable assertion checking.

We extended the syntax of expression with quantifiers and several logical operators that are required for writing more complete specifications. We also imported a set of mathematical symbols such as $\Rightarrow, \Leftarrow, \Leftrightarrow, \forall, \exists, \Sigma$ and Π into the language to represent boolean operators and quantifiers.

The behavioral specifications in Lime are used to describe the behavior of a class or method more precisely; they are also useful for dynamic run-time checks of specification violations. The resulting behavioral specifications are the resource for generating documentation.

The next quality factor that we improved is reusability. Reusability is a benefit often credited to the inheritance of object-oriented programming (OOP). Inheritance is a major feature of OOP; its basic idea is defining a new class as an extensions of existing classes.

Subtyping and subclassing [9, 47] are related but conceptually different views of inheritance: Subtyping is related to specification and interface inheritance; subclassing is a mechanism for implementation inheritance. The subtyping relation is in a behavioral hierarchy. The subclassing relation is in an implementation hierarchy.

The following examples show the difference between subclassing and subtyping.

```
class CartesianPoint
    attr x, y : real;
    method set (x, y : real)
```

```
    self.x, self.y := x, y
  method scale(s : real)
    x, y := x*s, y*s
end
class PolarPoint implement CartesianPoint
  attr r, phi : real;
  method set(x, y : real)
    self.r, self.phi := sqrt(x^2 * y^2), arctan(y/x)
  method scale(s : real)
    r := r * s
end

class Point
  attr x, y : integer;
end
class Line extend Point
  attr u, v : integer;
end
```

Class *PolarPoint* is defined as a subtype of class *CartesianPoint* using the clause *implement*. Class *PolarPoint* only inherits the interface of class *CartesianPoint*. It needs to provide implementation for the methods defined in class *PolarPoint*. Class *Line* is a subclass of class *Point*. While the *Line* class can use the *Point* class (to construct ifself), it is invalid to assign an instance of the class *Line* to a variable whose type is *Point*.

Currently, only a few languages, such as POOL-I [3], Theta [16], PolyTOIL [8] and Sather [56], support separating subtyping and subclassing. In strongly-typed object-oriented languages like Simula [7], Trellis [53], C++ [55], C# [36] and Java [22], subtyping and subclassing are combined and equated. In most such languages, subclassing is restricted to satisfy the requirements of subtyping. It has been argued that this eliminates several important opportunities for code reuse [31, 45].

We propose an inheritance mechanism for Lime, that separates subclassing from subtyping, and makes inheritance in Lime more flexible. Any class in Lime can act as a superclass or a supertype; it contains a specified interface, specified behavior, and implementation. A child class has the choice of inheriting either the behavioral interface specification, the implementation, or both.

Three different clauses are used for inheritance:

1. Subclassing uses the **extend** clause : *B extend A*

   Class *B* is called a subclass of class *A*, it can reuse the code in class *A*.

2. Subtyping uses the **implement** clause : *B implement A*

   Class *B* is called a subtype of class *A*, it inherits the interface and behavioral specification from class *A*.

3. Subclassing and subtyping often go along in parallel. For handling this common case, we use the **inherit** clause: *B inhert A*

   Class *B* inherits the behavioral interface specification and implementation from class *A*.

For type safety, we provide some means for dynamic run-time assertion checks that enhance subtyping to preserve the behavioral specifications inherited from all supertypes.

Other quality factors, such as extendibility, reusability, and maintainability, can benefit from automatic documentation generation. We developed a documentation tool called LimeD to support automatic documentation generation in Lime.

In traditional programming languages, the documentation is separated from its source code. The documentation may seem trivial, but many software projects are poorly documented [51]. When it becomes necessary to maintain a software project, the lack of documentation can be a major problem; and maintenance may cause the separated documentation to get out of sync with its source code. As software projects grow in size, software documentation becomes increasingly important. It is essential to keep documentation consistent and up-to-date.

Lime's support for automatic documentation generation was influenced by early work on literate programming. The term "Literate Programming" was coined by Knuth in his paper [27], in which also a particular documentation system called **WEB** was introduced. In WEB system, a tool called **tangle** can extract and assemble the program fragments according to the rules of the programming language. Another tool called **weave** formats the documentation, generates indexes, and presents all of it in a nice-looking paper format. The difference between the Lime documentation tool LimeD and most literate programming tools is that the goal of LimeD is to extract information from source code for documentation generation only. For this reason we do not have to extend our tool to support method implementations, as most literate programming tools do. For supporting inheritance, LimeD also needs to extract information from the class files of superclasses and supertypes.

The idea for the document layout was derived from *Javadoc* [19, 20, 29, 35], a tool from Sun Microsystems for generating API documentation in HTML format from documentation comments in Java source code. According to Lime features, the

document generated by LimeD contains information on the API and the behavior as specified by preconditions, postconditions and class invariants; it provides information on behavioral subtyping; it also displays the hierarchies of classes and types. The mathematical symbols for representing quantifiers and operators in a behavioral interface specification are properly displayed in a web browser.

My main contributions are making Lime a Behavioral Interface Specification Language (BISL) and augmenting Lime by following language features:

- Lime supports subclassing and subtyping as distinct features, making the inheritance mechanism flexible.
- Lime supports run-time checking of behavioral specifications.
- Lime supports automatic documentation generation with the aid of LimeD.
- Lime provides high readability and expressiveness of source code by supporting mathematical symbols.

1.3 Structure of the Thesis

The remainder of this thesis is organized as follows:

Chapter 2 gives an introduction on the work related to the added language features. Chapter 3 introduces the design of the added features. Chapter 4 gives the details of the implementation. Chapter 5 gives some testing results and applications. Chapter 6 draws the conclusions of our work.

Chapter 2

Related Work

For improving quality factors, we focus our work on specification and inheritance. The Lime design strategy follows the principles of object-orientation. The important feature of Lime is that it separates subtyping and subclassing. Lime also supports assertions, behavioral subtyping and automatic documentation generation. Some ideas are derived from Eiffel, JML and Literate Programming.

2.1 Assertions

Assertions are formal constrains on software system behavior that are commonly written as annotations of a source text. The primary goal in writing assertions is to specify what a system is supposed to do rather than how it does it. The use of assertion to specify software dates back to Hoare's 1969 paper on formal verification [23]. A correctness formula (also called Hoare triple) is an expression of the form

$$\{P\} \quad A \quad \{Q\}$$

where A denotes an operation, P and Q are properties of A, and P is called the precondition and Q is called the postcondition. It is a mathematical notation, not a programming construct, and used to express the properties of software elements.

Design by Contract (DBC), proposed by Meyer for Eiffel [45], is a formal technique for dynamically checking specification violation during runtime. The principal idea behind DBC is that a class and its client have a "contract" with each other. The client must guarantee certain conditions before calling a routine (method) defined by the class, and in return the class guarantees certain properties that will hold after the call. The contract describes under what conditions the software may be used and which task it is supposed to perform. These conditions and tasks can be considered benefits and obligations for a class and a client. An obligation for a class is a benefit for a client and vice versa.

- The precondition binds the client: it defines the conditions under which a call to the routine is legitimate. It is an obligation for the client and a benefit for the supplier.

- The postcondition binds the class: it defines the conditions that must be ensured by the routine on return. It is a benefit for the client and an obligation for the supplier.

The benefits are, for the client, the guarantee that certain properties will hold after the call; for the supplier, the guarantee that certain assumptions will be satisfied whenever the routine is called. The obligations are, for the client, to satisfy the requirements as stated by the precondition; for the supplier, to do the job as stated by the postcondition.

Preconditions and postconditions describe the properties of individual routines. There is also a need for expressing global properties of the instances of a class. A class invariant is such an assertion, specifying the allowed global states of a class and expressing restrictions on and relationships between the values of the attributes.

Meyer also proposes an invariant rule that precisely defines when an assertion is a correct invariant for a class.

An assertion I is a correct class invariant for a class C if and only if it meets the following two conditions:

- Every creation routine of C, when applied to arguments satisfying its precondition in a state where the attributes have their default values, yields a state satisfying I.
- Every exported routine of the class, when applied to arguments and a state satisfying both I and the routine's precondition, yields a state satisfying I.

Assertions have three major applications: they help produce reliable software, they provide systematic documentation, and they are a central tool for testing and debugging software.

2.2 Inheritance and Subtyping

Inheritance is a language mechanism that allows new object definitions to be based on existing ones. A new class inherits the properties of its parents, and may introduce new properties that extend, modify or defeat its inherited properties.

Inheritance is one of the characteristic features of object-oriented programming. It is often regarded as the feature that distinguishes object-oriented programming

from other programming paradigms.

Although the reason for distinguishing between inheritance and subtyping has been widely recognized and accepted, only a few programming languages, such as POOL-I [3], Theta [16], PolyTOIL [8] and Sather [56] separate subtyping from subclassing.

The following definitions have been proposed by LaLonde and Pugh [32].

- Subclassing is an implementation mechanism for sharing code and representation.
- Subtyping is a substitutability relationship; an instance of a subtype can stand in for an instance of its supertype.

Cook [13] points out that in most strongly-typed object-oriented languages, including C++, Eiffel, and Simula, types are equated with classes, and inheritance is basically restricted to satisfy the requirements of subtyping. He also suggests that subtyping should be clearly separated from subclassing.

America [2, 3] describes that in POOL-I subtyping is based on the externally observable behavior of objects. It includes not only their signature (the names of available methods and their parameter and result types) but also more detailed information about this behavior.

The essence of behavioral subtyping is summarized by Liskov and Wing's subtype requirement [39]:

> Let $\phi(x)$ be a property provable about objects x of the type T. Then $\phi(y)$ should be true for objects y of type S where S is a subtype of T.

Liskov and Wing [40, 41] also present a way of defining the subtype relation that ensures that subtype objects preserve behavioral properties of their supertypes. The subtype relation is based on the specifications of the sub- and supertypes.

Two kinds of properties of an object's behavior in a program must be preserved: invariants, which are properties true of all states, and history properties, which are properties true of all sequences of states.

A *type* defines a set of *values* for an object and a set of *methods* that provide the only means to manipulate that object. *Obj* is a set of unique identifiers for all objects that can contain values. A type is modeled as a triple $< O, V, M >$, where $O \subseteq Obj$ is a set of objects, $V \subseteq Val$ is a set of values, and M is a set of methods.

A type specification includes the following information:

- *The type's name;*
- *A description of the type's value space;*
- *A definition of the type's invariant and history properties;*
- *For each of the type's methods:*
  - *Its name;*
  - *Its signature;*
  - *Its behavior in terms of preconditions and postconditions.*

The behavioral subtype relation is rigorously described by Liskov and Wing whose definition is shown in Figure 2.1. In this definition they use the substitution notation

$$E[x/R]$$

to stand for expression E with all occurrence of the variable x replaced by expression R. Our Lime implementation follows the rules given in this definition.

Subtyping relation : $\sigma = < O_\sigma, S, M >$ is a subtype of $\tau = < O_\tau, T, N >$ written $\sigma \preceq \tau$ if there exists an abstraction function, $A : S \rightarrow T$, and a renaming map, $R : M \rightarrow N$, such that:

1. *Subtype methods preserve the supertype methods' behavior. If m_τ of τ is the corresponding renamed method m_σ of σ, the following rules must hold:*
   - *Signature rule.*
     - *Contravariance of arguments. m_τ and m_σ have the same number of arguments. If the list of argument types of m_τ is α_i and that of m_σ is β_i, then $\forall i \ . \ \alpha_i \preceq \beta_i$.*
     - *Covariance of result. Either both m_τ and m_σ have a result or neither has. If there is a result, let m_τ's result type be α and m_σ's be β. Then $\beta \preceq \alpha$.*
     - *Exception rule. The exceptions signaled by m_σ are contained in the set of exceptions signaled by m_τ.*
   - *Methods rule. For all $x : \sigma$:*
     - *Precondition rule. $m_\tau.pre[A(x_{pre})/x_{pre}] \Rightarrow m_\sigma.pre$.*
     - *Postcondition rule. $m_\sigma.post \Rightarrow m_\tau.post[A(x_{pre})/x_{pre}, A(x_{post})/x_{post}]$.*
2. *Subtypes preserve supertype properties. For all computations c and all states ρ and ψ in c such that ρ precedes ψ, for all $x : \sigma$:*
   - *Invariant Rule. Subtype invariants ensure supertype invariants.*

     $I_\sigma \Rightarrow I_\tau[A(x_\rho)/x_\rho]$
   - *Constraint Rule. Subtype constraints ensure supertype constraints.*

     $C_\sigma \Rightarrow C_\tau[A(x_\rho)/x_\rho, A(x_\psi)/x_\psi]$

Figure 2.1: Liskov and Wing's Subtype Relation Definition

2.3 Assertions and Inheritance in Programming Languages

2.3.1 APP, an Annotation PreProcessor for C

Rosenblum [52] describes an assertion processing system for C and UNIX called APP. APP provides a rich collection of features for specifying not only the assertions themselves but also the responses to failed runtime assertion checks.

APP recognizes assertions that appear as annotations of C source text. In particular, the assertions are written inside comment regions using the extended comment indicators /*@ ... @*/.

Each APP assertion specifies a constraint that applies to some state of a computation. The constraint is specified using C's expression language, with the C convention that an expression evaluating to zero is false, while a nonzero expression is true.

APP recognizes two enhancements to the C expression language within assertion regions: quantifiers and the operator **in**. Existential and universal quantification over finite domains can be specified using a syntax that is similar to C's syntax for **for** loops. The operator **in** can be used to indicate that an expression is be evaluated in the entry state of the function that encloses the expression.

APP recognizes four assertion constructs, each indicated by a different keyword:

- **assume** *specifies a precondition on a function;*
- **promise** *specifies a postcondition on a function;*
- **return** *specifies a constraint on the return value of a function;*
- **assert** *specifies a constraint on an intermediate state of a function body.*

The following examples illustrate the usage of assertion in APP. Since C is not an object-oriented language, inheritance issues are not a concern in APP. In function *swap*, the assumption states the precondition that the pointers x and y be non-null (and thus evaluate to true) and not equal to each other. The two postconditions use the operator *in* to relate the values of the integers upon exit from the function to their values upon entry. The specification of *sort* uses the existential quantifier *some* and universal quantifier *all* to state both the obvious ordering requirement of the result as well as the requirement that the result must be a permutation of the input array.

```
void swap (x, y)
int *x,  *y;
 /*@
   assume x && y && x != y;
   promise *x == in *y;
   promise *y == in *x;
@*/
{
...
}

int* sort(x, size)
int* x;
int size;
/*@
  assume x && size > 0;
  return S where S      // S is non-null
     && all (int i=0; i < in size-1; i=i+1) S[i] < S[i+1]  // S is ordered
     && all (int i=0; i < in size; i=i+1)
          some (int j=0; j < in size; j=j+1)
```

```
            x[i] == S[j];        // S is a permutation of x
@*/
{
...
}
```

2.3.2 Design by Contract in Eiffel

Eiffel is a landmark programming language that integrates executable specifications into the language. Design by contract is a way for developing reliable software. In Eiffel, the contracts are defined by program code, and are translated into executable code by the compiler. Thus, any violation of the contract can be detected immediately during runtime.

In Eiffel [44], *require* and *ensure* clauses are used to respectively introduce precondition and postcondtion. Each assertion is a list of boolean expressions, separated by semicolons; here a semicolon is equivalent to a boolean *and*. Boolean expressions in assertion have a few extensions such as the *old* notation. The notation *old e*, where e is an expression, denotes the value that e had on routine entry. Any occurrence of e not preceded by *old* in the postcondition denotes the value of the expression on exit.

A class invariant affects all the contracts between a routine of the class and a client. The correctness requirement on the routine may be expressed by the following correctness formula:

$$\{INV \wedge pre\}\ body\ \{INV \wedge post\}$$

Syntactically, a class invariant is an assertion, appearing in the *invariant* clause of the class, after the features and just before the *end*.

The following example shows assertions in Eiffel:

```
class BINARY_TREE [T]
    feature
        ... Attribute and routine declaration ...
    put_child(new: NODE) is
        -- Add new to the children of current node
        require
            new /= void
        do
          ... Insert algorithm ...
        ensure
            new.parent = Current
            child_count = old child_count + 1
         end  -- put_child
    invariant
         left /= Void implies (left.parent = Current)
         right /= Void implies (right.parent = Current)
end
```

The basic rules governing the rapport between inheritance and assertion have been proposed by Meyer [45]:

> **Parents' Invariant rule:** The invariants of all the parents of a class apply to the class itself.
>
> **Assertion Redeclaration rule (1):** A routine redeclaration may replace the original precondition by one equal or weaker, and the original postcondition by one equal or stronger.
>
> **Assertion Redeclaration rule (2):** In the redeclared version of a routine, it is not permitted to use a *require* or *ensure* clause. Instead one may:

- use a clause introduced by *require else*, to be or-ed with the original precondition.
- use a clause introduced by *ensure then*, to be and-ed with the original postcondition.

In the absence of such a clause, the original assertion is retained. Eiffel supports behavioral subtyping by following its parents's invariant rule and assertion redeclaration rules that imply Liskov and Wing's invariant rule, precondition rule and postcondition rule.

2.3.3 Java Specification with JML

The lack of assertions and design by contract features for checking of preconditions, postconditions, and invariants in the Java language has led to some languages and run-time assertion checking tools for Java, such as Alloy Annotation Language (AAL) [26], Jass [6], and iContract [30]. AAL is a language for annotating Java code based on the Alloy modeling language. It offers a syntax similar to JML and has the same opportunities for generation of run-time assertions. JML, which stands for "Java Modeling Language," is a behavioral interface specification language (BISL) designed to specify Java modules.

As an example, class *Purse* illustrates JML's main features. JML adds assertions to Java by writing them as special comments (/*@ ... @*/ or //@ ...). Within such comments JML extends the Java syntax with several keywords—in this example, *invariant, requires, assignable, ensures*, and *signals*. It also supports three forms of quantifier expressions: universal and existential quantifiers (*forall* and *exists*), generalized quantifiers (*sum*, *product*, *min*, and *max*, and a numeric quantifier

(*num_of*).

The central ingredients of a JML specification are preconditions (given in *requires* clauses), postconditions (given in *ensures* clauses), and class invariants. In addition to "normal" postconditions, JML also supports "exceptional" postconditions, specified in *signals* clauses. The *signals* clause in the class *Purse* specifies that method *debit* may throw a *PurseException*, and, in that case, the balance will not change (as specified by the use of the \old keyword). The *assignable* clause for the method *debit* specifies a frame condition, namely that *debit* will assign to only the balance field.

```
public class Purse  {
     final int MAX_BALANCE;
     int balance;
     //@ invariant 0 <= balance && balance <= MAX_BALANCE;

     byte[] pin;
     /*@ invariant pin != null && pin.length == 4
       @    && (\forall int i; 0 <= i && i < 4; 0 <= pin[i] && pin[i] <= 9);
       @*/

     /*@ requires amount >= 0;
       @  assignable balance;
       @  ensures    balance == \old(balance) - amount && \result == balance;
       @  signals  (PurseException) balance == \old(balance);
       @*/
       int debit(int amount) throws PurseException {
           if (amount <= balance) { balance -= amount; return balance: }
           else { throw new PurseException("overdrawn by " + amount); }
       }
```

```
    /*@ requires   0 < mb && 0 <= b && b <= mb && p != null && p.length == 4
      @            && (\forall int i; 0 <= i && i < 4; 0 <= pin[i] && pin[i] <= 9);
      @ assignable MAX_BALANCE, balance, pin;
      @ ensures    MAX_BALANCE == mb && balance == b
      @            && (\forall int i; 0 <= i && i < 4; p[i] == pin[i]);
      @*/
   Purse(int mb, int b, byte[] p) {
        MAX_BALANCE = mb;  balance = b; pin = (byte[])p.clone();
   }
}
```

JML adapts some notation such as $\backslash old$ and $\backslash result$ to Eiffel. A difference from Eiffel is that JML extends the syntax of Java expressions with quantifiers and other constructs that are needed for logical expressiveness. Without supporting quantifiers, Eiffel has to use an auxiliary method call in assertions to express and to check the same content as a quantifier does.

In JML, a subclass inherits specifications such as preconditions, postconditions, and invariants from its superclasses and interfaces that it implements. An important feature of JML's specification inheritance is that its semantics supports a behavioral notion of subtyping. JML checker handles JML's specification inheritance [17]. That is, it combines specifications from superclasses and superinterfaces and use as those to form a complete specification of a class. In essence, preconditions are disjoined, postconditions are conjoined in the form of $\bigwedge(\backslash old(p_i) \Rightarrow q_i)$ (where p_i is a precondition and q_i is the corresponding postcondition), and invariants are conjoined. That meets Liskov and Wing's substitution property.

2.3.4 Subtyping and Subclassing in Sather, Theta and POOL-I

Sather [48, 56] is an object-oriented language developed at the International Computer Science Institute. It has statically-checked strong (contravariant) typing, multiple inheritance, separate implementation and type inheritance, exception handling, assertions, preconditions, postconditions, and class invariants.

Sather provides separate mechanisms for subtyping and subclassing concepts. *Abstract classes* represent interfaces: sets of signatures that subtypes of the abstract class must provide. Other kinds of classes provide implementations. Classes may include an implementation from other classes using a special *include* clause; this does not affect the subtyping relationship between classes.

In the following examples, *$PLAYER* is an abstract type. The name of abstract type must be in capital letters. The leading $ differentiates abstract from concrete types. The abstract type *$PLAYER* specifies the common interface. In the body of the type declaration, the operations are given without any implementation.

Class *PLAYER* is a concrete type or class which is a subtype of *$PLAYER*. The subtype relation is expressed by the < symbol. Class *PLAYER* will not be used for instantiation; there will be no objects of type *PLAYER*. The main purpose of this class is to declare attributes and routines that are common to other classes of type *$PLAYER*, which *include* the implementation of this class. The routine *getmove* does not provide a basic implementation. However, for consistency with the interface required by *$PLAYER*, a dummy implementation must by given. The routine *ask_pawn_xchg* provides a default implementation. Concrete class *HUMAN* is a subtype of *$PLAYER*. It provides an implementation for at least the signatures given in the specification of

$PLAYER. *HUMAN* inherits the implementation of class *PLAYER* by the *include* statement, and it redefines routine *getmove* and *ask_pawn_xchg*.

```
type $PLAYER is
    getmove(b: BOARD): MOVE:
    ask_pawn_xchg: CHAR:
end;

class PLAYER < $PLAYER is
    attr iswhite: BOOL;
    create(iswhite: BOOL): SAME is
       ret: SAME := new;
       ret.iswhite := iswhite;
       return ret;
    end;
    getmove(b: BOARD): MOVE is
       raise "PLAYER: invalid call to getmove\n";
    end;
    ask_pawn_xchg: CHAR is
       return 'Q';
    end;
end;  --of class PLAYER

class HUMAN < $PLAYER is
  include PLAYER;
  getmove(b: BOARD): MOVE is
    return MAIN::display.getmove(iswhite);
  end;
  ask_pawn_xchg: CHAR is
       MAIN::display.update(MAIN::board.str);
```

```
      return MAIN::display.ask_pawn_xchg;
   end;
end; -- of class HUMAN
```

Sather supports separating subtyping and subclassing, but there exist some restrictions for Sather's types and classes. Only abstract classes such as *$PLAYER* can have descendants in the type hierarchy, but cannot be instantiated as objects. Concrete classes, such as *PLAYER* and *HUMAN*, are always leaf nodes in the subtype graph, but can be instantiated. All Sather variables are statically typed. If a variable is declared as a concrete type, then only objects of exactly that type can be held by it. As a result, all calls on such variables are monomorphic, i.e. the actual implementation invoked is statically determined.

In Sather, abstract classes only provide interfaces, a set of signatures, no behavioral specification; statements for precondition, postcondition and invariant are used in concrete classes to check the correctness. Sather's subtype relation only follows the signature rule defined in Liskov and Wing's subtype relation, that is the contra/covariance rules. It does not support behavioral subtyping.

Theta [16] is an object-oriented language that was developed for use within the Thor object-oriented database system. A Theta program consists of a group of program units. A program unit is a specification or a module. Specifications define abstract types and routines; modules provide implementations of these types and routines. New types and routines are defined by providing specifications. A specification defines interface information; it does not include any information about how the new type or routine is to be implemented. The specification of a type defines the supertypes, and the names and signatures of the methods. A type can have multiple

supertypes. Classes and routine implementations are placed inside modules. Each module implements one or more specifications. A type is implemented by one or more classes. A class contains some instance variables that store the state of objects of the type, and routines that implement the object's methods. A class can inherit from a single superclass. Objects of a subclass contain superclass instance variables and methods.

In the following examples, type *bag* gives a simple abstraction. Type *stack* is defined as a subtype of *bag* using '<'. The specification of *stack* introduces a new method, *top*. It contains specifications for *push* and *pop*, since they constrain the behavior of the corresponding *bag* methods, and for *copy*, since it has a different signature and specification than its counterpart, but it omits the specification of *size*, since it would be identical to its counterpart. Class *brep* implements type *bag* and provides an access to the array that contains its elements, e.g., by providing a *get_els* method, which returns the *els* component of a bag. Class *srep* implements type *stack* and inherit *brep*. *stack* implementation is dependent on the details of the *bag* implementation, e.g., that *put* adds the new element to the high end of the array and *get* removes the newest element.

```
bag = type [T]
     put (x: T)
     get ( ) returns (T) signals (empty)
     size ( ) returns (int)
     copy ( ) returns (bag[T])
         where T has copy( ) returns (T)
end bag

stack = type [T] < bag[T] {push for put, pop for get}
```

```
    push (x: T)
    pop ( ) returns (T) signals (empty)
    top ( ) returns (T) signals (empty)
    copy ( ) returns (stack[T])
        where T has copy ( ) returns (T)
end stack

brep = class[T] for bag[T]
    ...
    sz: int implements size   % implementation of size method
    els: array[T] implements get_els % implementation of get_els method
    put (x: T)
        els.append(x)
        sz := sz + 1
        end put
    get ( ) returns (T) signals (empty)
        x: T := els.remove( ) except when bounds: signal empty end
        sz := sz - 1
        return (x)
        end get
    copy ( ) returns (bag[T])
        where T has copy ( ) returns (T)
        return (brep[T]{sz := sz, els := els.copy( )})
        end copy
 end brep

 srep = class[T] for stack[T] inherits brep[T] {push for put, pop for get}
     top ( ) returns (T) signals (empty)
        return (self.get_els( ).top( ))
             except when bounds: signal empty end
```

```
        end top
    copy ( ) returns (stack[T])
        where T has copy ( ) returns (T)
        return (srep[T]{mk_copy[T](self)})
        end copy
end srep
```

Theta provides separate mechanisms for type hierarchy and inheritance. For a strongly-typed object-oriented language, the compiler needs to know the type hierarchy: which types are subtypes of which other types. In Theta this information is provided in type specifications. The inheritance mechanism is separate from the type hierarchy, so that related types can have independent implementations and unrelated types can have related implementations.

Theta does not support assertions. The subtype relation only follows the signature rule defined in Liskov and Wing's subtype relation.

The language POOL-I [2, 3] is a member of the POOL family of languages, which have been designed to support the development of large programs for highly parallel machines with or without shared memory. POOL-I is the first language that includes subtyping and inheritance as completely separate language mechanisms. In POOL-I subtyping is based only on the externally observable behavior of objects. This includes not only their signature but also more detailed information about the behavior.

In the following examples, the specifications of integer bag, stack, and queue are given by type *Int_Bag*, *Int_Stack*, and *Int_Queue* respectively. The class *AIS* implements the type *Int_Stack*, because it has the required methods *get* and *put* with the right argument and result types and the property LIFO.

```
TYPE Int_Bag
METHOD get () : Int
METHOD put (Int) : Int_Bag
END Int_Bag

TYPE Int_Stack
PROPERTY LIFO %% last in, first out
METHOD get () : Int
METHOD put (Int) : Int_Stack
END Int_Stack

TYPE Int_Queue
PROPERTY FIFO %% first in, first out
METHOD get () : Int
METHOD put (Int) : Int_Queue
END Int_Queue

CLASS AIS
VAR a := Array(Int).new(1, 0)
METHOD get () : Int
BEGIN IF a@ub = 0
      THEN RESULT NIL
      ELSE RESULT a@high %% also decreses ub
      FI
END get
METHOD put (n: Int) : AIS
BEGIN IF a@high : = n;  %% increase ub and place n at high and
      RESULT SELF
END put
PROPERTY LIFO
```

```
END AIS
```

The relations among user-defined types are not declared explicitly in POOL-I. The type mechanism in POOL-I offers the possibility to analyze types dynamically. Since every object that is an element of the type *Int_Stack* or *Int_Queue* will certainly have the methods *get* and *put*, it will also be an element of the type *Int_Bag*. Therefore, the types *Int_Stack* and *Int_Queue* are subtypes of the type *Int_Bag*.

The Subtyping in POOL-I depends not only on the signatures of the different types (the names of the methods and their parameter and result types) but also on the behavior. The specification of a type in POOL-I is augmented with a collection of properties, which are just identifiers. The compiler will not check whether the code of a class really satisfies such a specification; it will just base its decisions about subtyping relationships on the presence or absence of these property identifiers. The types *Int_Stack* and *Int_Queue* have different properties, so they are different types and neither of them is a subtype of the other one.

The POOL-I makes a distinction between types and classes: A type is a collection of objects that have the same behavior; a class is a collection of objects that look exactly the same on the inside. We say that a class *implements* a type σ when each object of the class has type σ. There may be many classes that implement a certain type and many types that are implemented by a certain class. If a class implements a type, it automatically implements all the supertypes of this type. The class *AIS* implements the type *Int_Stack*, because it has the required methods *get* and *put* with the right argument and result types and the property LIFO. The class *AIS* also implements the type *Int_Bag* that is a supertype of *Int_Stack*.

2.4 Mathematical Symbol Representation

Solving mathematical problem and evaluating boolean expressions is the fundamental usage of programming languages, but the most elementary mathematical symbols such as $\leq, \geq$, and $\neq$ are not used in programming languages. Languages such as Pascal use $<=, >=$ and $<>$ to stand for $\leq, \geq$ and $\neq$. That is different from the way they are printed in books or any other printed source on paper.

Microsoft's Equation Editor and Math Type are popular mathematical equation editors. Equations exist as graphic objects such as OLE (Object Linking and Embedding), EGO (Edit Graphic Object) in the documentation of Microsoft's Equation Editor and Math Type; and they can be saved as EPS (Encapsulated PostScript) files, WMF files (Windows Metafiles), or PICT files (standard Macintosh graphics). This way is not suitable for programming language source code that has to be text or extended text as required for compilation parsing.

Numerous symbolic mathematical operators are used in formal specification languages such as VDM [25] and Z [58, 15]. They use LaTeX to present mathematical formulae and symbols. It requires the user of Z to have knowledge of LaTeX; and the source code on which the programmer works is in the LaTeX format that can be parsed, but is not very suitable for reading.

Currently, most text editors such as TextEdit, jEdit, AbiWord and Microsoft Word support Unicode encoding such as UTF-8 and UTF-16. With the aid of Unicode encoding and its corresponding font, mathematical symbols are properly represented in the above text editors and are suitable for parsing. Though Unicode cannot handle complicated mathematical formula, it is enough for us to present the mathematical symbols used in our language.

Unicode [1, 18] is the international standard whose goal is to specify a code matching every character needed by every written human language to a single unique integer number, called a code point. Unicode has become the dominant encoding scheme in internationalization of software and multilingual environments. Windows OS such as Windows NT, Windows 2000 and Windows XP make extensive use of Unicode, more specifically UTF-16, as an internal representation of text. UNIX-like operating Systems such as Linux, BSD and Mac OS X have adopted Unicode, more specifically UTF-8, as the basis of representation of multilingual text.

UTF-8 (8-bit Unicode Transformation Format) is a lossless, variable-length character encoding for Unicode. A Unicode character's bits are divided into several groups, which are then divided among the lower bit positions inside the UTF-8 bytes. The contents of Table 2.1 are from Lindholm and Yellin's book [37]. All characters in the range '000000' to '00007F' are represented by a single byte. The seven bits of data in the byte give the value of the character represented. Characters in the range '000080' to '0007FF' are represented by a pair of bytes a and b, the bytes represent the character with the value $((a \ \& \ 0x1f) << 6) + (b \ \& \ 0x3f)$. Characters in the range '000800' to '00FFFF' are represented by three bytes a, b, and c. The character with the value $((a \ \& \ 0xf) << 12) + ((b \ \& \ 0x3f) << 6) + (c \ \& \ 0x3f)$ is represented

Code range	**UTF-8 binary**	Value (a: 1st byte, b: 2nd byte, c: 3rd byte)
000000 - 00007F	0xxxxxxx	a
000080 - 0007FF	110xxxxx 10xxxxxx	$((a \ \& \ 0x1f) << 6) + (b \ \& \ 0x3f)$
000800 - 00FFFF	1110xxxx 10xxxxxx 10xxxxxx	$((a \ \& \ 0xf) << 12) + ((b \ \& \ 0x3f) << 6) + (c \ \& \ 0x3f)$

Table 2.1: UTF-8 Encoding

by the bytes. The Unicode and UTF-8 encoding for the mathematical symbols used in Lime is given later on in Table 3.1.

2.5 Automatic Documentation Generation Tools

Program documentation is a very useful and essential resource for programmers and system users, however producing and keeping it up-to-date can be an expensive and time-consuming endeavor. Many tools have been developed to address this issue.

A traditional computer program consists of a text file containing program code. Scattered in amongst the program code are comments that describe the various parts of the code. Donald Knuth changes the attitude to the construction of programs in his literate programming paradigm [27]:

> Instead of imaging that our main task is to instruct a computer what to do, let us concentrate rather on explaining to human beings what we want a computer to do.

Knuth's approach is to combine Pascal code with TEX documentation to produce a new language WEB, that offers programmers a superior approach to programming. The idea is that a programmer writes one document, the web file, that combines documentation (written in TEX) with code (written in Pascal).

Running on the web file, the WEB system produces a complete Pascal program, ready for compilation by an ordinary Pascal compiler and a TEX file, ready to be processed by TEX. The resulting documents include a variety of automatically generated indices and cross-references that make it much easy to navigate the code.

Since then a number of researchers have worked sporadically on the ideas as well as on the tools for literate programming. The **CWEB** [28] system of Structured

Documentation is a version of WEB for documenting C, C++, and Java programs. Thus CWEB combines TEX with today's most widely used professional programming languages. **FWEB** [5] is a WEB system of structured documentation for multiple languages. **Noweb** [24] is designed to meet the needs of literate programmers while remaining as simple as possible. It has been used for tens of thousands of lines of code in such languages as C, C++, Haskell, Modula-3, Objective Caml, and Standard ML.

One of the main ideas behind literate programming is to formulate the understanding of the program, and connect this understanding to traditional program source. Literate programming has attracted interest in the functional programming community. Since a short functional language program can contain more information and is more difficult to understand, recording the understanding is more important. There are three tools (**IDoc, HDoc** and **Haddock** [43]) supporting Haskell documentation.

A different kind of documentation system, such as **Javadoc** and **Doxygen** have only little to do with the ideals of literate programming; They generate on-line interface documentation in HTML format.

Doxygen can also generate off-line reference manual (in LATEX or RTF) from a set of source files. Doxygen can be configured to extract the code-structure from undocumented source files. This can be very useful to quickly find your way in large source distributions.

The reuse of library components is a cornerstone of today's software development. Therefore, developers need good library documentation to take advantage of complex APIs. The major source of Java API information consists of Web pages generated by the **Javadoc** tool [19, 20, 29, 35]. The Javadoc program extracts the names, parameters, return types, and other necessary information from classes and methods with the aid of a Java parser. The developers can also augment their source code by writ-

ing special tagged comments. Javadoc then uses these comment tags to strengthen the documentation.

Lime supports automatic documentation generation using LimeD. The development of LimeD got ideas from these documentation tools discussed above. The design and goals for developing LimeD is explained in Chapter 3.

Chapter 3

Design Features of Lime

In this chapter, we discuss the analysis and design of the new Lime features. We research the reason and background of the added features and introduce them in Chapter 2. In Chapter 1, we have introduced the added Lime language features. Here, we discuss them in further detail. We also describe the details for Lime on how to handle adding new features.

3.1 Assertion

We should design Lime to integrate assertions as programming language constructs, as Eiffel does. We should take the *old* and *result* notation and the idea that assertions are written using Lime's expression syntax from Eiffel. To allow us to easily write complete specifications, we extend the syntax of Lime expressions with quantifiers and other constructs that are needed for logical expressiveness and are not supported by Eiffel. Assertion constructs should meet the requirements for specifying the behavior of a Lime module, providing information for automatic documentation and for being

executable for dynamically checking specification violation.

An assertion is a predicate which states a logical sentence that evaluates to true or false. If during the program execution the assertion evaluates to false, an error is indicated. Lime uses the following three kinds of assertions to specify its module behavior:

- class invariant,
- method precondition, and
- method postcondition.

In addition, Lime has assert statements that specify a constraint on an intermediate state of a method body. The assert statement starts with the keyword *assert*.

A class invariant specifies the allowed global states of a class. The class invariant is declared with the keyword *invariant* and it is located between the declarations of attribute and method. The class invariant is checked whenever the state of the class is *stable*; i.e., it is checked at the beginning and end of each public method's execution.

A precondition can be used to specify the valid states for method invocation. The precondition is checked at the beginning of the method body. It is declared with the keyword *pre*.

A postcondition specifies the legal states after method invocation. The postcondition is checked at all normal return points of the methods; i.e., it is checked for all return statements and at the end of method bodies. The postcondition is declared with the keyword *post*.

To emphasize the behavioral specification, both the precondition and the postcondition are located at the beginning of the method body. They will be reorganized according to their execution order in the generated intermediate language.

3.2 Mathematical Symbols

Formal specification languages such as VDM and Z make heavy use of symbolic mathematical operators, which discourages use by programmers. These kind of formal specification languages focus on formal theorem proving and reasoning; they do not support runtime assertion checking. Languages, like Eiffel limit assertions to expressions of the underlying programming language, which makes it difficult to write complete specifications. JML and APP extend boolean expression with quantifiers that are executable for runtime specification violation checking. They represent quantifiers with special keywords composed of ASCII characters. For example, universal quantifier and existential quantifier are represented by *forall* and *exists* in JML, *all* and *some* in APP respectively. They are not very meaningful. Especially when specification appears in automatically generated documentation whose readers are not programmers in the underlying programming language, these keywords may be confusing.

We should design Lime to moderately use mathematical symbols for quantifiers and operators. This will make formal specification more meaningful, and it will make the language source code and generated documents highly readable and unambiguous. The specification should be executable for formal verification. Lime should be more expressive for specification than Eiffel, easier to use than VDM and more meaningful and readable than JML and APP.

To make the source code and generated documentation more readable and meaningful, we import a number of mathematical symbols shown in Table 3.1 to Lime language. With the aid of Unicode and UTF-8 encoding, these mathematical symbols can be parsed by the Lime compiler and displayed in, for example,TextEdit

for editing source code and inside generated documentation on web browsers. Some symbols are used for replacing old operators; others are used in extended expressions.

Symbol	Name	Unicode	UTF-8 binary	Note
≤	lessequal	2264	**1110**0010 **10**001001 **10**100100	replace <=
≥	greaterequal	2265	**1110**0010 **10**001001 **10**100101	replace <=
≠	notequal	2260	**1110**0010 **10**001001 **10**100000	replace <>
∧	logicaland	2227	**1110**0010 **10**001000 **10**100111	logical and
∨	logicalor	2228	**1110**0010 **10**001000 **10**101000	logical or
¬	logicalnot	00AC	**110**00010 **10**101100	negation
⇒	arrowdblright	21D2	**1110**0010 **10**000111 **10**010010	implication
⇐	arrowdblleft	21D0	**1110**0010 **10**000111 **10**010000	consequence
⇔	arrowdblboth	21D4	**1110**0010 **10**000111 **10**010100	if and only if
∀	universal	2200	**1110**0010 **10**001000 **10**000000	universal quantifier
∃	existential	2203	**1110**0010 **10**001000 **10**000011	existential quantifier
Σ	summation	2211	**1110**0010 **10**001000 **10**010001	sum
Π	product	220F	**1110**0010 **10**001000 **10**001111	product

Table 3.1: Mathematical Symbols Used in Lime

3.3 Expression Extensions

To have more expressiveness that makes more specialized assertion languages convenient for writing behavioral specifications, Lime extends its expressions with various constructs, such as quantifiers.

Lime adds the following new constructs to its expression syntax:

- ⇒ for logical implication. It is the only one binary infix operator that associates

to the right. The expressions on either side of these operators must be type of *boolean*, and the type of the result is also *boolean*.

- $\Leftarrow$ for logical consequence. The expressions on either side of these operators must be type of *boolean*, and the type of the result is also *boolean*.

- $\Leftrightarrow$ for logical equivalence. The expressions on either side of these operators must be type of *boolean*, and the type of the result is also *boolean*. Note that $\Leftrightarrow$ means the same things as $=$ for expressions of type *boolean*; however, $\Leftrightarrow$ has a lower precedence.

- *old* for referring to value in the pre-state. It is used in *post* clause to indicate an expression whose value is taken from the pre-state of a method call. For example, *old e* denotes the value of the expression e evaluated in the pre-state of a method call.

- *result* for referring to the value or object that is being returned by a method. It is used in a method's *post* clause. Its type is the return type of the method.

- The general linear quantifier of the form $* \ x \ | P \bullet E$, where $*$ is a quantifier operator; x is the bound variable of the quantification; P is the range of the quantification; E is an expression, the body of the quantification. Seven quantifiers are added.

  - $\forall$ and $\exists$ are universal and existential quantifiers respectively. Their type is *boolean*. If the range predicate P evaluates to *false*, i.e. an empty range, the value of universal quantification defaults to *true*; the value of existential quantification defaults to *false*.

– Σ, Π, MAX and MIN are generalized quantifiers that return the sum, product, maximum or minimum of the values of the expressions given, where the variable satisfies the given range. The expression in the body must be of numeric type, which currently is *integer* only. The type of the quantified expression as a whole is the type of its body. The range predicate must be of type *boolean*. For empty range, Σ has default value 0; Π has default value 1; MAX and MIN have no default value, they will indicate an error.

– NUM is a numerical quantifier. It returns the number of values for its variables for which the range and the expression in its body are true. The entire quantifier expression has type *integer*. It has default value 0. Both the range predicate and the body must have type *boolean*.

The precedence of operators in expressions is given in Figure 3.1. Operators on the first line have the highest precedence, on the lowest line have the lowest precedence, on the same line have the same precedence. The Lime language guarantees that the operands of operators appear to be evaluated in a specific evaluation order, namely, from left to right.

Figure 3.2 shows the syntax of the extended expressions in Lime. We use the following notation to describe the syntax. A nonterminal symbol is denoted by italic font; a terminal symbol is put between two double quote marks; the extended parts are denoted by bold font.

- $M \mid N$ denotes that alternative between M and N (either M or N);
- MN denotes that the concatenation of M and N (M followed by N);

	$\forall$ $\exists$ Σ Π MAX MIN NUM () method calls
	$(unary) -$ $+$ $\neg$ *not*
	$*$ / *div* *mod*
	$(binary) +$ $-$
	$<$ $>$ $\leq$ $\geq$ $=$ $\neq$
	$\wedge$ *and*
	$\vee$ *or*
	$\Rightarrow$ $\Leftarrow$
	$\Leftrightarrow$
lowest	:=

Figure 3.1: Precedence of Operators

- $[M]$ denotes that M is optional;
- $\{M\}$ denotes a sequence of M (the sequence may be empty);
- (MN) denotes that M and N are grouped, i.e. $(M|N)M$ means MM or NM.

3.4 Separate Subclassing and Subtyping

First of all Lime is an object-oriented programming language, it should support OOP's characteristic features — encapsulation, polymorphism and especially inheritance.

We should design Lime to support separate subclassing and subtyping. We also should design Lime to make inheritance more flexible so that any class can be a superclass and supertype that gives the rights to child classes to choose what (method signature, behavioral specification or implementation) is to be inherited from the parent class.

Expression ::= ***EquivalenceExpr***

EquivalenceExpr ::= ***ImplicationExpr*** {"⇔" ***ImplicationExpr***}

ImplicationExpr ::= ***ConditionalOrExpr*** ["⇒" ***ConditionalOrExpr***
| "⇐" ***ConditionalOrExpr***]

ConditionalOrExpr ::= *ConditionalAndExpr* {"∨" ***ConditionalAndExpr***
| "or" *ConditionalAndExpr*}

ConditionalAndExpr ::= *RelationalExpr* {"∧" ***RelationalExpr***
| "and" *RelationalExpr*}

RelationalExpr ::= *AdditiveExpr* ["<" *AdditiveExpr* | ">" *AdditiveExpr* | "≤" *AdditiveExpr*
| "≥" *AdditiveExpr* | "=" *AdditiveExpr* | "≠" *AdditiveExpr*]

AdditiveExpr ::= *MultiplicativeExpr* {"+" *MultiplicativeExpr*
| "−" *MultiplicativeExpr*}

MultiplicativeExpr ::= *UnaryExpr* {"∗" *UnaryExpr* | "/" *UnaryExpr* | "div" *UnaryExpr*
| "mod" *UnaryExpr*}

UnaryExpr ::= "−" *UnaryExpr* | "¬" *UnaryExpr* | "not" *UnaryExpr*
| "+" *UnaryExpr* | ***PrimaryExpr***

PrimaryExpr ::= *Literal* | *Ident* | "(" *Expression* ")"
| ***AssertionPrimary*** | ***Call***

AssertionPrimary ::= ***Old*** | ***Result*** | ***SpecQuantifiedExpr***

Old ::= "old" (***Ident*** | "(" ***Expression*** ")")

Result ::= "result"

SpecQuantifiedExpr ::= ***Quantifier Ident*** "|" ***AdditiveExpr*** ("<" | "≤") ***Ident***
("<" | "≤") ***AdditiveExpr*** "•" ***Expression***

Quantifier ::= "∀" | "∃" | "Σ" | "Π" | "MAX" | "MIN" | "NUM"

Figure 3.2: Syntax of Lime Expressions

τ *extend* σ

1. τ inherits attribute a_σ of σ: $\tau.A \supseteq \sigma.A$.
2. For each non-private method m_σ of σ there is a corresponding method m_τ of τ, such that
   - m_τ gets m_σ's signature: $m_\tau.Sig = m_\sigma.Sig$.
   - m_τ has m_σ's implementation: $m_\tau.Imp = m_\sigma.Imp$.

Figure 3.3: Definition of *extend*

Lime has three access modifiers, *private, protected* and *public*. We should design Lime to allow using private attributes in specifications such that the private attributes can be accessed in a subtype's inherited specification.

A Lime class definition consists of a class invariant (I), a set of attributes (A) and a set of methods (M). We model a class as a triple $\langle I, A, M\rangle$. A method is composed of a signature (Sig), behavioral specification and implementation (Imp). The Method signature includes name, access, return and parameters' types. The behavioral specification consists of a precondition (Pre) and a postcondition ($Post$). The implementation is the source code of the method body. We model a method as a quadruple $\langle Sig, Pre, Post, Imp\rangle$.

Lime uses the *extend* clause to handle single subclassing (shown in Figure 3.3); *implement* clause to handle multisubtyping (shown in Figure 3.4); *inherit* clause to handle the common case of subclassing and subtyping (shown in Figure 3.5).

We use the following examples to show the usage of the inheritance clauses. The method *sum1to9* in the class A calculates the sum of sequence 1, 2, ..., 9.

τ *implement* $\sigma_1, \sigma_2, ..., \sigma_n$

1. τ preserves invariants of all supertypes $(\sigma_1, \sigma_2, ..., \sigma_n)$: $\tau.I \Rightarrow \bigwedge_{i=1}^{n} \sigma_i.I$.
2. τ inherits all attributes from all supertypes $(\sigma_1, \sigma_2, ..., \sigma_n)$: $\tau.A \supseteq \bigcup_{i=1}^{n} \sigma_i.A$.
3. For each non-private method m_{σ_i} of each supertype σ_i there is a corresponding method m_τ of τ, such that
   - m_τ gets m_{σ_i}'s signature $(m_\tau.Sig = m_{\sigma_i}.Sig)$.
   - m_τ weakens preconditions: $m_\tau.Pre \Leftarrow \bigvee_{i=1}^{n}(m_{\sigma_i} \in \sigma_i.M \wedge m_{\sigma_i}.Pre)$.
   - m_τ strengthens postconditions: $m_\tau.Post \Rightarrow \bigwedge_{i=1}^{n}(m_{\sigma_i} \in \sigma_i.M \Rightarrow m_{\sigma_i}.Post)$.

Figure 3.4: Definition of *implement*

```
public class A
    public attr sum : integer;
    private attr i : integer;
    invariant sum ≥ 0
    public method sum1to9: integer
        pre sum = 0
        post result = ∑ i | 0<i <10 • i
        begin
            sum := (1+9)*9/2;
            println(sum);
            return sum
    end
```

τ *inherit* σ (equivalent to τ *extend* σ *implement* σ)

1. τ preserves invariant of σ: $\tau.I \Rightarrow \sigma.I$.
2. τ inherits attribute a_σ of σ: $\tau.A \supseteq \sigma.A$.
3. For each non-private method m_σ of σ there is a corresponding method m_τ of τ, such that
   - m_τ gets m_σ's signature: $m_\tau.Sig = m_\sigma.Sig$.
   - m_τ has m_σ's implementation: $m_\tau.Imp = m_\sigma.Imp$.
   - m_τ weakens preconditions: $m_\tau.Pre \Leftarrow m_\sigma.Pre$.
   - m_τ strengthens postconditions: $m_\tau.Post \Rightarrow m_\sigma.Post$.

Figure 3.5: Definition of *inherit*

```
    initialization
        sum := 0
end
```

When we define a new class B as: *class B extend A*, class B will inherit attributes *sum* and *i*, and method *sum1to9* with implementation from A. Lime allows to override the inherited method. Class B can redefine method *sum1to9* to return any integer value, that will not cause any problem, since class B only inherits the method's signature and implementation not behavioral specification.

```
class B extend A
    public method sum1to9: integer \\ override inherited method
```

```
        return 0 \\ OK since no assertion required
end
```

For case *class B implement A*, class B inherits attributes *sum* and i, and behavioral interface specification that includes class invariant, method's preconditions and postconditions, and method's signatures without implementation. Class B has to provide an implementation for method *sum1to9*, otherwise class B should be declared as *abstract*. The implementation can be different, but must preserve the inherited behavioral specifications that will be dynamically checked at runtime.

```
class B implement A
    public method sum1to9: integer
        return 45 \\ make it satisfy inherited postcondition (result = ∑ i | 0<i <10 • i)
end
```

For case *class B inherit A*, class B inherits attributes *sum* and i, and the behavioral interface specification that includes class invariant, method's preconditions and postconditions, and the method's signatures with implementation.

3.5 Inheritance of Specification

Inheritance is very powerful, but is dangerous too. Without the possibility of client control, class developers can use redeclaration and dynamic binding to change the semantics of operations treacherously. In most OOP Language such as Java and C++, being the subtyping relation only requires involved classes to conform the contra/covariant rules that only ensure that no type errors happen.

Lime as a BISL should meet the subtyping requirement proposed by Liskov and Wing [39]. We should design Lime to carry out checking of the following rules defined in Liskov and Wing's definition of subtype relation [40]:

- *Precondition rule* ensures the subtype's method can be called at least in any state required by supertype.

- *Postconditon rule* says that the subtype method's postcondition can be stronger than the supertype method's postcondition.

- *Invariant rule* states subtype invariants ensures supertype invariants.

The precondition rule ensures the subtype's method can be called at least in any state required by the supertype. The postcondition rule says that the subtype method's postcondition can be stronger than the supertype method's postcondition; hence, any property that can be proved based on the supertype method's postcondition also follows from the subtype method's postcondition. The invariant rule shows that a subtype's invariant implies a supertype's invariant.

In Lime, a subtype inherits the specification such as preconditions, postconditions and invariants from its supertypes. An important language feature of Lime is that its semantics supports a behavioral notion of subtyping. To ensure that a program continues to work as expected in situations where a supertype object is replaced by a subtype object, Lime supports the standard *design by contract* weakening of preconditions and strengthening of postcondition.

When Lime performs a runtime specification violation checking in its subtype, preconditions and postconditions of a method are combined with the corresponding preconditions and postconditions of the supertypes. Preconditions are combined with

a logical *or* and postcondition with a logical *and* thus achieving a weakening of preconditions and a strengthening of postconditions.

In addition, the class invariant of a subtype is combined with invariants inherited from all its supertypes by using a logical *and*, and checked in every stable state, i.e. at the end of the initialization and the beginning and end of each non-private method's execution.

The approach for dynamically checking assertions in Lime makes it follow the precondition rule, postcondition rule and invariant rule in the Liskov and Wing's subtype definition.

3.6 Documentation Generation

The essence of and related work in automatic documentation generation is given in Chapters 1 and 2. We should design Lime to automatically generate documentation from source code. For this purpose, we propose a documentation tool, LimeD.

The core goals for developing LimeD are

- to generate documentation directly from source code;
- to provide a behavioral interface specification, not only an application programming interface (API);
- to show the hierarchies of classes and types;
- to make the resulting documents be hyperlinked, highly readable, and easily navigated.

The design of LimeD is focused on achieving the goals that make LimeD generate interface and specification documentation.

The document must be automatically generated from the source code. A static document is too likely to get out of sync with the source code on which it is based.

Some data structures in the Lime compiler hold the information that is necessary for generating documentation. For example, we can extract information on attributes and methods from symbol tables. We built our documentation tool LimeD based on the existing Lime compiler.

We add documentation comments and documentation tags to Lime syntax for documentation generation purpose. LimeD supports embedded HTML inside the documentation comment. LimeD not only provides Lime API but also provides behavioral specification informations such as class invariant, method preconditions and postconditions. For supporting behavioral subtyping, LimeD provides all supertype's behavioral specification that are extracted from their class files.

LimeD shows the inheritance relations by different means: the class hierarchy is shown as a graphic, subtype hierarchy is shown as an indented list.

To make the documentation easily navigated and to support cross references, we make full use of the Web's capabilities by generating documentation in HTML format so that it can be presented in various web browsers. LimeD generates an all class list with links and leaves an anchor in every class and method's name that makes navigating inside or between documentations easy.

Currently, LimeD can generate interface and specification documentation. One future work on LimeD would be extending to provide system design documentation that requires to collect all information on aggregation, interaction, and inheritance between the classes inside the system. Aggregation information can be obtained from the non-primitive type attribute declaration. Interaction information can be obtained from the method invocation. The current version of LimeD already has a

way to get inheritance information. All the information obtained has to be held by an appropriate data structure and presented graphically.

Chapter 4

Implementation

In this chapter, we discuss the details of the implementation of the Lime compiler and LimeD. We introduce the structure of the software and the algorithms that are used for adding new language features. We also introduce some solutions in more details. Before doing that, we first introduce our compilation strategy and tools used in the compiler.

4.1 Background

The parser is generated by Java Compiler Compiler (**JavaCC**). The Lime compiler first translates its source code, in which the character set is extended to **Unicode** for presenting mathematical symbols, into an intermediate **Jasmin** [46] code that uses the Java Virtual Machine (**JVM**) [37] instructions set; during this step, specification clauses are reorganized according to their execution order. Jasmin, a Java assembler, converts intermediate language code into byte-code **Java class files**. For supporting inheritance and separate compilation, the compiler extracts information from Java

class files with the aid of the Byte Code Engineering Library (**BCEL**).

4.1.1 JavaCC

Sun's JavaCC is a parser generator and lexical analyzer generator. JavaCC will read a language's description written in a LEX/YACC-like [4] manner and generate code, written in Java, that will read and analyze that language.

A JavaCC grammar is specified using code-like extended BNF. Both the lexical and grammar specification are contained in the same file. JavaCC also provides lexical state and lexical action capabilities.

JavaCC is a DFA-based parser generator. It generates recursive-descent parsers. A recursive-descent parser is a top-down parser built from a set of mutually recursive procedures, each of which implements a grammar production rule. By default, JavaCC generates an LL(1) parser. However, there may be portions of the grammar that are not LL(1). For example, classes and methods have modifiers such as *abstract, private*, and *public*. JavaCC offers the capabilities of syntactic and semantic lookahead to resolve shift-shift ambiguities locally at these points.

JavaCC also provides the capability of building an Abstract Syntax Tree (AST) via a tool called JJTree. JJTree is a pre-processor for JavaCC that inserts parse tree building actions at the appropriate places in the JavaCC grammar source. The output of JJTree is run through JavaCC to create the parser. By default, JJTree generates code to build parse tree nodes for each nonterminal in the language. Although JavaCC is a top-down parser, JJTree constructs the parse tree from the bottom up. JavaCC does not generate output languages. However once the AST has been generated, it is easy to generate code from it.

4.1.2 Java Virtual Machine

A virtual machine is a software emulation of a real machine. It has an instruction set and manipulates various memory areas at run time just like a physical machine.

The JVM's instruction set is composed of byte-codes. A byte-code is either a byte-sized instruction or a byte-sized operand. When a Java source code file is compiled, the Java compiler converts program statements to byte-codes. The compiler creates a *class* file to contain the output of byte-codes. When the Java runtime system runs a Java program, the data contained in the class files is loaded into memory and Java interpreter is started. The Java interpreter reads each successive byte-code and passes the instruction and operands to the JVM.

The JVM [37] consists of an instruction set, a set of registers and an operand stack. The registers of the JVM maintain machine state during its operation. They are directly analogous to the registers of a microprocessor. The JVM's registers include:

- PC: the Java program counter;
- VARS: a pointer to the first local variable of the currently executing method, all local variables are addressed relative to this register;
- OPTOP: a pointer to the top of the Java operand stack;
- FRAME: a pointer to the execution environment of the currently executing method.

JVM is a stack-based machine. On each Java method invocation, the JVM allocates a Java frame, which contains an operand stack. Most JVM instructions take values from the operand stack of the current frame, operate on them, and return

results to the same operand stack. The operand stack is also used to pass arguments to methods and receive method results.

4.1.3 Format of Java *Class* Files

The Lime compiler converts a Lime source code to a Java *class* file. For supporting separate compilation, we need to extract information from Java *class* files. In this part, we introduce the format of Java *class* file. The contents are taken from [11, 37] with some modification.

The *class* file structure is as follows where type *u1 and u2* represent an unsigned one-byte and two-byte quantity, respectively.

```
ClassFile {
    cp_info constant_pool[];
    u2 access_flags;
    u2 this_class;
    u2 super_class;
    u2 interfaces_count;
    u2 interfaces[interfaces_count];
    u2 fields_count;
    field_info fields[fields_count];
    u2 methods_count;
    method_info methods[methods_count];
}
```

The attributes in the *ClassFile* structure are as follows:

- *constant_pool*: It is an array structure where each entry has a *cp_info* sub-

structure. It contains string constants that represents the name of the classes, interfaces, fields, methods and other constants that are referenced within the *ClassFile* structure and its substructures.

- *access_flags*: It denotes the access permissions and other modifier set up of a class.

- *this_class*: It is a valid index into the *constant_pool* array, and the entry at that index is a *CONSTANT_Class_info* structure representing the class or interface defined by this *class* file.

- *super_class*: If it is zero, then this *class* file represents the class *Object*. Otherwise, it is a valid index into the *constant_pool* array, and the entry at that index is a *CONSTANT_Class_info* structure representing the direct superclass of the class defined by this *class* file.

- *interfaces_count*: It denotes the number of the interfaces of the given class.

- *interfaces[]*: It contains a set of valid indices into the *constant_pool* array, and the entry at each index is a *CONSTANT_Class_info* structure representing the direct superinterface of the class defined by this *class* file.

- *fields_count*: It denotes the number of *field_info* structures of a given class.

- *fields[]*: It contains a set of *field_info* structures that gives a complete description of a field that is declared in the class defined by this *class* file.

- *methods_count*: It denotes the number of *method_info* structures of a given class.

- *methods[]*: It contains a set of *method_info* structures that gives a complete description of all methods that are declared in the class defined by this *class* file.

Every entry in the constant pool is a *cp_info* structure:

cp_info {

u1 tag;

u1 info[];

}

The value of attribute *tag* determines the type of the string constant this *constant_pool* entry represents. Attribute *info[]* contains corresponding information about the string constant.

The *CONSTANT_Class_info* structure is used to represent a class or an interface. It has the following format:

CONSTANT_Class_info {

u1 tag;

u2 name_index;

}

The attribute *tag* has value 7. The *name_index* is a valid index into *constant_pool*, and the entry at that index is the name of the class or interface represented by this *CONSTANT_Class_info* structure.

Fields and methods that are referenced in a class are entries in the *constant_pool* of that class represented by the following structures:

CONSTANT_Fieldref_info {

u1 tag;

u2 class_index;

u2 name_and_type_index;

}

CONSTANT_Methodref_info {

u1 tag;

u2 class_index;

u2 name_and_type_index;

}

The *tag* has the value 9 or 10 in *CONSTANT_Fieldref_info* or *CONSTANT_Methodref_info*, respectively. The value of *class_index* is a valid index into *constant_pool*, and the entry at that index is a *CONSTANT_Class_info* structure that represents the class or interface in which the field or method is declared. The *name_and_type_index* is a valid index into *constant_pool*, and the entry at that index represents the name and type of the field or the name, return type and argument type of a method, respectively.

Every field or method declared in a class is described by a *field_info* or *method_info* structure, respectively. They have the following format:

field_info {

u2 access_flags;

u2 name_and_type_index;

u2 attributes_count;

attribute_info attributes[attributes_count];

}

method_info {

u2 access_flags;

u2 name_and_type_index;

```
}
attribute_info {
    u2 attribute_name_index;
    u4 attribute_length;
    u1 info[attribute_length];
}
```

The value of the *access_flags* denotes the access permission and other modifier set up of the field or method. The *name_and_type_index* is a valid index into *constant_pool*, and the entry at that index represents the name and type of the field or the name, return type and argument type of a method, respectively.

4.1.4 Jasmin

Jasmin is a Java assembler developed by Jon Meyer and Troy Downing [46]. It takes a textual description of a class, written in a simple assembler-like syntax using the JVM instruction set and converts this into a binary Java *class* file suitable for loading into a Java interpreter.

Lime first converts its source code to Jasmin assembler code; Jasmin converts it to a Java class file. This avoids getting into the details of constant pool indices, attribute tables and so on.

The following listing shows the fragments of the Jasmin file that is generated from the Lime example used in Section 3.4.

```
.class public A_C[1]
.super java/lang/Object
.implements A
.field public sum I
.field protected i I
.method public sum1to9() I
    .limit stack 4
    .limit locals 5
     ...
     aload_0
     iconst_1
     bipush 9
     iadd
     bipush 9
     imul
     iconst_2
     idiv
     putfield A_C/sum I
     getstatic java/lang/System/out Ljava/io/PrintStream;
     aload_0
     getfield A_C/sum I
     invokevirtual java/io/PrintStream/println(I)V
     ...
     ireturn
.end method
```

The above example shows Jasmin's syntax. We can take a look at the instructions and directives that start with a "." character used in this example. The first three lines contain information about the class defined in the file. Three directives are used in this part. The *.class* and *.super* directive tell the JVM the name of this class and its

[1]The reason for suffix "_C" will be explained later.

superclass. After *.class* and *.super*, it is a list of the interfaces that are implemented by the class you are defining, using the *.implements* directive.

After the header information, the next section of the Jasmin file is a list of field definitions.

A field is defined using the *.field* directive:

.field <access-spec> <field-name> <descriptor> [= <value>]

The capital "I" in the *.field* directive is a type descriptor. Table 4.1. shows the type descriptors used in Jasmin.

Descriptor	**Type**	**Meaning**
B	byte	signed byte
C	char	charactor
D	double	double-precision IEEE float
F	float	single-precision IEEE float
I	int	integer
J	long	long integer
S	short	signed short
Z	boolean	true or false
[C	char[]	single-dimensional array of 16-bit Unicode chars
[[F	float[][]	two-dimensional array of floats
[Ljava/lang/Thread	Thread[]	single-dimensional array of threads
V	void	method returns no result

Table 4.1: Type Descriptors in Jasmin

After listing the fields of the class, the rest of the Jasmin file lists methods defined by the class.

A method is defined using the basic form:

.method <access-spec> <method-spec>

<statements>

.end method

The following directives can be used only within method definitions:

.limit stack <integer>

sets the maximum size of the operand stack required by the method.

.limit locals <integer>

sets the number of local variables required by the method.

JVM instructions are placed between the *.method* and *.end method* directives. JVM instructions can take zero or more parameters, depending on the type of instruction used.

4.1.5 Byte Code Engineering Library

For generating byte codes Java *class* file, we get help from Jasmin; we also need to extract information from *class* files for supporting separate compilation and inheritance. BCEL is a good helper for doing this work.

BCEL operates at the level of actual JVM instructions; so it can let you dig into the bytecode of Java classes and give you a convenient possibility to analyze, create, and manipulate Java *class* files. BCEL parses the Java *class* file and generates a set of objects which contain all the symbolic information of the given class: methods, fields and byte code instructions, in particular.

4.2 Strategy of Separating Subtyping and Subclassing

Lime uses three clauses (*extend, implement* and *inherit*) to establish the inheritance relation. For example, the class header

class Sub extend Sup2 implement Sup1

builds an inheritance relation shown by an inheritance graph in Figure 4.1. In this graph, as well as in all other Lime inheritance graphs, solid and dashed arcs are used to represent subtype and subclass relationships, respectively.

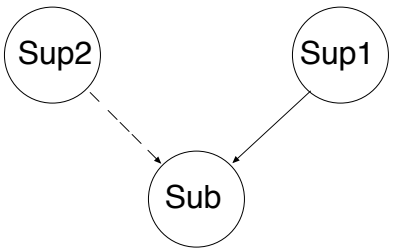

Figure 4.1: Inheritance Graph in Lime View

Now, we discuss how the inheritance relationship is implemented in the generated executable Java *class* file. Java supports single inheritance and implementation of multiple interfaces that can only contain method signatures and constant static variables. The superclass and the interfaces are referred to by *super_class* and *interfaces[]* in *ClassFile* structure. For each Lime source class, we generate two Java *class* files. One stores a Java class that contains all the information in the original Lime file, and

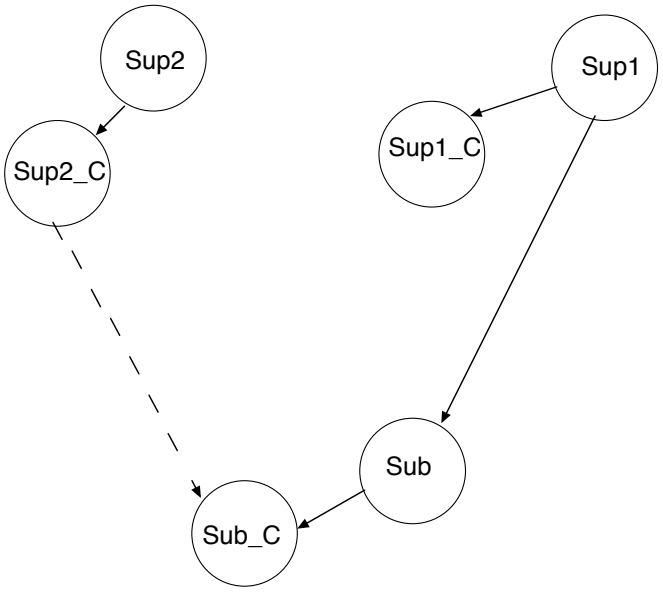

Figure 4.2: Inheritance Graph of Java Classes

its name ends with "_*C*", the other stores a Java interface that still uses its original name. An attribute declaration such as

attr s : Sub;

is translated to a Jasmin code like

.field s LSub

that generates the same content in Java *class* file as we declare it in Java as:

Sub s;

When we create an instance in Lime, we have to create it from a class type. For example, the statement

s := new Sub();

will be translated to

s := new Sub_C()

in the background. For any Lime class X, we also make the generated class X_C be a subtype of type X by using a directive (*.implement* X) in X_C's Jasmin code so that the above assignment statement meets the requirement of the Lime type checking system.

The graph in Figure 4.2 shows the inheritance relationship among the generated Java classes. For Java, it is legal to assign an instance of *Sub_C* to a variable declared as a type of *Sup1* or *Sup2*. According to our definition of *extend*, *implement* and *inherit*, class *Sub* is *Sup1*'s subclass, not subtype. We built a type checking system that checks whether the assigned variable type is the supertype of the instance's class. From the inheritance graph view, this job is to check whether there exists a path that is composed of all solid arcs between two types. Since a supertype does not contain any information on its subtypes, a subtype holds its direct supertypes in its *interfaces[]* that builds a subtype relationship in a directed acyclic graph (DAG) that may have multiple levels. Checking system travels the subtype relation DAG and searches for the assigned supertype. If it exists in the subtype DAG, then the substitution is valid.

From the inheritance graph, we can see that the subclass (*Sub_C*) is still a subclass of superclass (*Sup2_C*) in the generated Java classes inheritance relationship, so it can inherit attributes, method's interfaces and implementation from the superclass. The substitution is prohibited by the Lime type checking system; the subtype (*Sub*) extends its supertype interface *Sup1*, so it can inherit method interfaces from its supertypes. We discuss how to handle inheriting specified behavior from supertypes in Section 4.4.

4.3 Handling Inheritance

In this section, we discuss how to handle inheritance in Lime. We only focus on handling method inheritance. We use similar ways to handle attribute inheritance. The following system is used for illustrating the process of handling method inheritance. This system consists of the following classes:

class C inherit S13 implement S11, S12

class S11 extend S23 implement S21, S22

class S12 implement S24

class S13 extend S25

class S25 extend S32 implement S31

class S21

Classes *S22, S23, S24, S31* and *S32* are leaf node classes, they have similar class declaration as class *S21*. The inheritance relation graph is shown in Figure 4.3.

Clause *implement* establishes a subtype relation represented as a solid arc. Clause *extend* establishes a subclass relation represented as a dashed arc. Clause *inherit* handles a kind of relation in which the child class is a subtype also a subclass of its parent class; this kind of relation is represented as two parallel arcs, one is solid, the other is dashed. This kind of relation exists between class *C* and class *S13*.

Handling method inheritance involves a number of Java classes such as *JavaClassHandler, ClassParser, JavaClass, MethodInfoGenerator, MethodInfo* and *MethodTable*. Classes *ClassParser* and *JavaClass* are from BCEL. The Lime parser contains a method table of type *MethodTable* to hold the information on all methods that are defined in current class or inherited from superclasses and supertypes. When the Lime parser handles inheritance clauses (*extend, implement* and *inherit*),

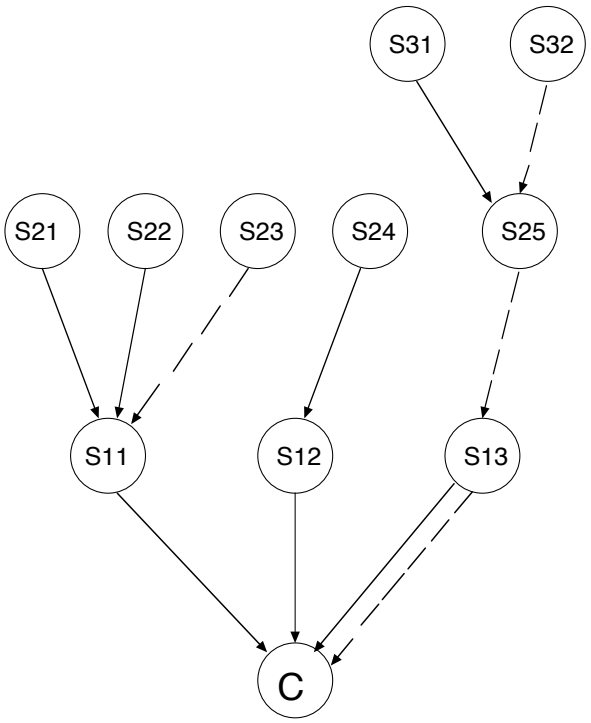

Figure 4.3: Inheritance Graph of the Example System

it traverses the inheritance DAG in the depth first search (DFS) order, the order for our example system is:

$$S21 \longrightarrow S22 \longrightarrow S23 \longrightarrow S11 \longrightarrow S24 \longrightarrow S12 \longrightarrow S31 \longrightarrow S32 \longrightarrow S25 \longrightarrow S13 \longrightarrow S31 \longrightarrow S32 \longrightarrow S25 \longrightarrow C.$$

For each class node, *JavaClass* extracts method's information from Java *class* file corresponding to current node. *MethodInfoGenerator* generates a set of instance of *MethodInfo* that are added to the method table. The structure of *MethodInfo* is as follows:

```
MethodInfo {
    int accessflag;
    String className;
    String methodName;
    String methodArgument;
    String methodReturn;
    String methodPre = "";
    String methodPost = "";
}
```

This process is briefly represented by the sequence diagram shown in Figure 4.4.

Traversing in the inheritance DAG is achieved by recursively calling class *JavaClassHandler*'s method:

public static void handleInheritance(String supFilePath, boolean isInterface, boolean isSupertype, boolean beAbstract, boolean isInherit, MethodTable mT, int inheritLevel)

This method is placed in Appendix A. The parameters, *isInterface, isSupertype, beAbstract,* and *inInherit*, are used for tracing and handling different inheritance cases:

- Case 1: class node reaches current class node via all dashed arcs. This kind of classes such as *S13, S25* and *S32* are pure superclasses. The current class inherits method's interface and implementation from them. Non-*abstract* methods in these classes are kept non-*abstract*.

- Case 2: class node reaches current class node via all solid arcs. This kind of classes such as *S21, S22, S11, S24, S12* and *S13* are pure supertypes. Current class inherits method's interface and behavioral specification that is precondition and postcondition from them. All methods in these classes are generated

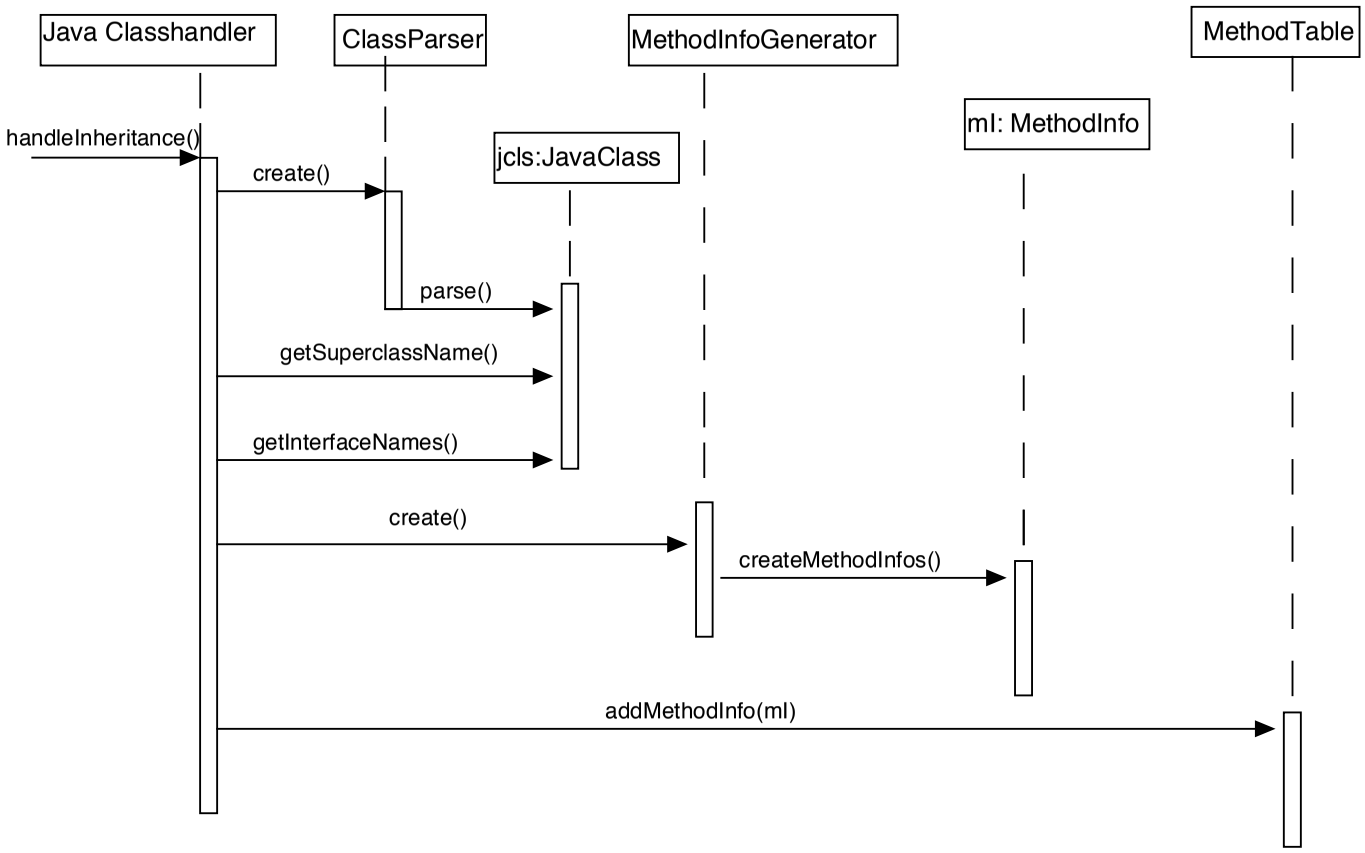

Figure 4.4: Sequence Diagram for Classes Handling Method Inheritance

as *abstract MethodInfo* by setting its *accessflag*, that means current class does not inherit their implementation.

- Case 3: class node reaches current class node via mixed (dashed and solid) arcs. This kind of classes such as *S23* and *S31* act as Java interface. Current class can only inherit the method interface from them. Methods in these classes are generated as *abstract MethodInfo* that contains no specification.

Class *MethodTable*'s method

public void addMethodInfo (MethodInfo mI)

whose source code is attached in Appendix A, also needs to handle different cases:

- Case 1: the added *MethodInfo* does not exist in method table. Just add it.

- Case 2: the class node of the adding *MethodInfo* is an ancestor node of the class node of the existing *MethodInfo*. Override the *accessflag*, reset *className* and handle behavioral specification.

- Case 3: the class nodes of the adding *MethodInfo* and the existing *MethodInfo* are in a different branch. Merge behavioral specification, only when the adding *MethodInfo* is non-abstract and the existing one is abstract, override its *accessflag* as non-abstract and reset *className*.

During handling inheritance, some fundamental language issues, such as abstract class instantiation, overriding method with weaker access privileges, are handled.

4.4 Supporting Assertion and Behavioral Subtyping

Lime supports *assertions* and uses *pre, post* and *invariant* clauses to formulate behavior specifications. The grammar for the *assert* statement is as follows:

Assert ::= "assert" expression

After an exception mechanism is provided to Lime, *assert* will be implemented as follows:

if ¬ expression then raise an exception

We currently implement it as follows: If the expression does not hold, then print assertion failure message and call system *exit* to halt execution.

Clauses *pre, post* and *invariant* are used to respectively introduce method precondition, postcondition and class invariant. For emphasizing the behavioral specification, clause *invariant* is defined at the beginning of the class, just following the attribute declaration; clauses *pre* and *post* are defined at the beginning of a method. They are not in the execution position. We have to reorganize them according to their execution order. During code generation, we respectively place *pre* and *post* statements at the beginning and the end of the method in which they are defined so that *pre* and *post* can be checked before and after method execution respectively; we also generate a *checkInvariant* statement for the class invariant and place it at the end of every method and *initialization* in the class. Then the class invariant will be checked at every stable state.

Statements, *pre, post* and *checkInvariant*, are implemented in the same way as the *assert* statement. Their failure message contains their corresponding information such as the name of the method and class in which they are defined.

Lime supports behavioral subtyping. Since Java does not support behavioral specification, we have to find a way to store the information of preconditions, postconditions and invariants in a Java *class* file. When we handle inheritance and separate compilation, we always get the information of the other compilation units from their Java *class* file. The reason is that in some situation such as using a library class, we may not have the source code.

The class invariants, preconditions and postconditions are stored in *field[]* as *field_info*; invariant, preconditions and postconditions are respectively named by *classInvariant*, method name concatenated with "pre" and method name concatenated with "post". Their values, that is the predicate expression, are stored in *attribute[]* as *attribute_info* so that they can be extracted as constant values from the constant pool.

Lime handles behavioral specification inheritance by traversing the subtype relation DAG and collecting all behavioral specifications in supertypes. For the situation shown in Figure 4.3, current class C collects behavioral specification from class *S21, S22, S11, S24, S12* and *S13*. After the invariant, preconditions and postconditions are extracted from the Java *class* files, the invariants are combined with logical *and*, "$\wedge$"; the preconditions of the methods with the same signature are combined with logical *or*, $\vee$; the postconditions of the methods with the same signature are combined with logical *and*, $\wedge$. The combined preconditions and postconditions are kept in the method table; the class invariant is assigned to its a corresponding variable. Statements *pre, post* and *checkInvariant* in the method generated code use the combined expression instead of the expression defined in the current class so that dynamic assertion checking can ensure that subtype preserves the behavior defined in all its supertypes.

4.5 Extending Lime Expression

Lime adds a number of new constructs to its expression syntax for expressiveness. Most of the new added constructs involves boolean operations. At the JVM level, there is no boolean type. Instead, booleans are represented as integers: 0 is used for *false*; and 1 for *true*. Now, we discuss the new constructs' implementation.

Logical implication, $a \Rightarrow b$, could be implemented as

$a \leq b$

The above implementation is fine, but it is not a lazy evaluation that is used for the existing logical operations *and* as well as *or*. Lazy evaluation is a concept that attempts to minimize the work the computer has to do. It has two related, yet different, meanings that could be described as delayed evaluation and minimal evaluation. The evaluation strategy that we use is minimal evaluation in which an expression is only evaluated until the point where its final value is known. This means that in some cases it's not necessary to evaluate all the parts of an expression. We implement it as "$\neg a \vee b$", that is

if $\neg a$ *then true else* b

This implementation can handle undefinedness of b in case that a evaluates to *false*. This is because a evaluates to false, $\neg a$ will be true, and true $\vee$ b evaluates to true for any value of b.

Similarly, logical consequence, $a \Leftarrow b$, is implemented as

if a then true else $\neg b$

Logical equivalence, $\Leftrightarrow$ has the same implementation as "=", but it is given a lower precedence by the Lime grammar.

Two notations, *result* and *old*, can be used in postconditions. Keyword *result* in a

method's postcondition holds the method's return value. Its implementation sequence is: first we need to declare a variable with the method return type and named by the method name concatenated with "result"; then we change the implementation of the *return* statement, let it assign the return value to the special variable, then jump to *checkInvariant* statement; the implementation of *result* is referring to the special variable.

The *old* construct in postconditions is used for referring to the pre-state value of its expression. The notation *old* is more difficult to implement than *result*, since each method postcondition can have a number of *old* constructs associated with different expressions. During parsing, we count each *old* construct and give it a serial number; we also pass the token list, which is generated for the *post* expression, to its corresponding method's AST node in which we do the related code generation. We declare variables for each *old* construct. During code generation, we get the expression of the *old* construct from the token list of *post* expression and replace its tokens by a new token with the corresponding variable so that in the new *post* expression generated from the exchanged token list, the *old* construct has been replaced by this variable. We put an assignment statement, that assigns the extracted expression to the corresponding variable, in the beginning of the method. Figure 4.5 demonstrates the above implementation.

Every quantification is based on an operator such as $+, *, \wedge, \vee$. We use the following pseudocode to show the implementation. Quantifications are directly translated to Jasmin code in the actual implementation.

$\forall\ i \mid low \leq i \leq up \bullet exp(i)$ is implemented as

for i:= low to up do

if $\neg$*exp(i) then return false;*

Original Lime source code

invariant I_{cur}

method m : T

pre exp_{pre}

post exp_{post} // including *result* and *old*(e_i) ... *old*(e_{i+n})

begin

...

return a;

end

Generated code

attr mresult : T ;

attr old_i ...

invariant I_{cur}

method m : T

begin

old_i := *old*(e_i);

...

pre $exp_{pre} \vee exp_{preInh}$; // exp_{preInh} inherited from all supertypes

begin // the start of original method body

...

mresult := a;

end; //end of original method body

checkInvariant $I_{cur} \wedge I_{Inh}$; // I_{Inh} inherited from all supertypes

post $exp_{post} \wedge exp_{postInh}$; // $(exp_{post} \wedge exp_{postInh})[result, old(e_i), ..., old(e_{i+n}) \backslash mresult, old_i, ..., old_{i+n}]$

end // end of method

Figure 4.5: Demonstration of Implementaiton for *old* and *result*

return true;

$\exists\ i \mid low \leq i \leq up \bullet exp(i)$ is implemented as

for i:= low to up do

if exp(i) then return true;

return false;

$\sum i \mid low \leq i \leq up \bullet exp(i)$ is implemented as

for (sum := 0, i:= low to up do)

sum := sum + exp(i);

return sum;

$\Pi\ i \mid low \leq i \leq up \bullet exp(i)$ is implemented as

for (product := 1, i:= low to up do)

*product := product * exp(i);*

return product;

$MAX\ i \mid low \leq i \leq up \bullet exp(i)$ is implemented as

if (low > up) then error;

for (max := exp(low), i:= low to up do)

if exp(i) > max then

max := exp(i);

return max;

$MIN\ i \mid low \leq i \leq up \bullet exp(i)$ is implemented as

if (low > up) then error;

for (min := exp(low), i:= low to up do)

if exp(i) < min then

min := exp(i);

return min;

$NUM\ i \mid low \leq i \leq up \bullet exp(i)$ is implemented as

for (num := 0, i:= low to up do)

if exp(i) then

num++;

return num;

4.6 Implementing LimeD

For each project, LimeD generates a sub-directory *docs* to hold all generated documentation files. To unify the documentation style, LimeD generates a Cascading Style Sheets (CSS) file. LimeD also generates a summary page for the whole project, that contains all classes' declarations. For easily accessing class documentation, a frame linked indices for all classes is generated and acts as a navigation menu. The main page in a frame view consists of two frames, the left frame contains the navigation menu, the right frame contains the summary page.

The documentation of each individual class starts with the class description extracted from the documentation comment in the source file. Documentation com-

ments can contain embedded HTML code. The document may contain the following parts:

- **Class Invariant** contains the invariant defined in the current class and the invariants inherited from all supertypes. The inherited invariants are "anded" to generate a single expression. It also gives the invariant that should be preserved by the current class. All the information is extracted with the aid of the Lime parser that holds the current class invariant and gets inherited ones from Java *class* files of supertypes.

- **Class Hierarchy** Lime supports single subclassing. The source code contains first level superclass name in *inherit* or *extend* clause. The upper level superclass name has to be extracted from its direct subclass's Java *class* file. We use a recursive function to extract all level superclass and display the class hierarchy graphically.

- **Type Hierarchy** We can use the same strategy for superclass to get information on supertypes. Since Lime supports multiple subtypes, we use an indented list to present subtype relation.

- **Attribute** contains all non-private attributes defined in the current class.

- **Inherited Attribute** contains all attributes inherited from superclasses and supertypes. All inherited attributes are extracted from Lime parser's symbol table.

- **Method** contains all methods defined in the current class. It provides not only the method signature but also precondition and postcondition defined in the

current class. It also provides the preserved assertions: if the method redefines or implements the one in supertypes, it has to preserve the precondition and postcondition defined in supertypes.

- **Inherited Method** contains all inherited methods. It provides the method signature and preserved precondition and postconditions.

In all parts mentioned above and in individual methods, we leave a named anchor for navigating between pages and inside a page. A normal way to display mathematical symbols on web browser is to use decimal reference or entity reference in HTML, for example, using Σ or &*Sigma*; to display Σ. Since Lime uses UTF-8 encoding for storing its source code, the behavioral specification obtained from the Lime parser is still in UTF-8 encoding. We do not need to translate mathematical symbol to its decimal or entity reference. We can choose the character set in generated HTML by adding the following meta between HEAD.

<meta http-equiv="Content-Type" content="text/html; charset=utf-8" >

Chapter 5

Testing

To verify the design and implementation of the added Lime features, we performed testing after each developing stage. The examples used are small, simple, and just for testing purposes. They may not be very meaningful.

5.1 Fundamental Test

While adding the inheritance mechanism to Lime, we also had to handle many related issues. We use the examples shown in Figure 5.1 to test the following cases. The Lime source code and test results are attached in Appendix B.

- Assigning a value to a constant variable: For an assignment statement in class A

  z := 2; // z is declared as a const

  Lime compiler should be able to detect it and produce an error message.

- Non-abstract class declaration: Class C is declared as

  class C implement B

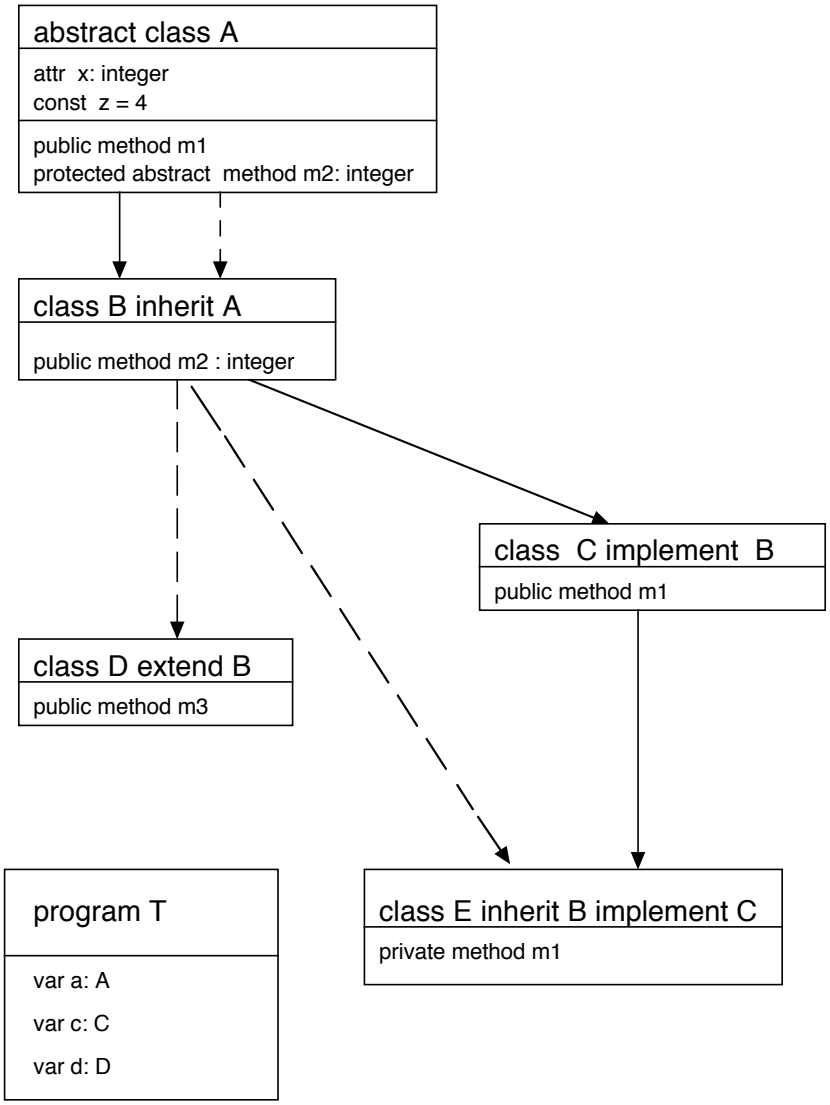

Figure 5.1: Fundamental Test Examples

then C inherits the interfaces of method *m1* and *m2* from class B. It only implements *m1*, so method *m2* is still an *abstract* method. Class C cannot be declared as a non-abstract class. The Lime compiler should give an error message.

- Assigning weaker access privilege: Class B overrides a *protected* method *m2* declared in class A as *public*, which is a legal method overriding. Class E tries to override a *public* method *m1* as *private*, which is an illegal method overriding. Class E is a subtype of C, B and A, method *m2* in supertypes can be used in a larger scope, any one can call it. After a substitution of an object of class C by an object of class E, the method *m1* cannot be accessed in the original scope. The Lime compiler should give an error message for this case.

- Creating an abstract class instance: In program T, the first line

  a := new A();

  tries to create an instance of the abstract class A. The Lime compiler should detect it and leave an error message.

- Assigning incompatible type: Assume following lines in Program T

  a := d;

  a is a variable declared as type A; d is a variable as type D and holds an instance of class D. Class D extends class B, it is not a subtype of A. The Lime compiler should prohibit this kind substitution and give an error message.

- Showing polymorphism in Lime: Lime supports polymorphism using dynamic linking. In the following lines of program T

  a := new B();

a.m1;

a := new E();

a.m1;

the first invocation of method *m1* should execute the method *m1* defined in class B, the second should execute the one in class E.

The test results show that the Lime compiler can handle the above issues, and satisfies the design requirements.

5.2 Assertion and Extension Expression Test

We use class AS to test the sequence of dynamic assertion checking and the handling of undefinedness in implication and consequence expressions implemented by using lazy evaluation.

class AS
  attr x: integer;
  attr y: integer;
  attr i: integer;
  attr res: integer;
  invariant x = 3
  public method dbcTest
    pre x = y
    post y = res
    begin
      x := 3;
      y := 5;
      assert x > y;
      res := $\Sigma\ i|0 \leq i \leq 6 \bullet i + 1$;
      println(res);

```
        res := Π i|0 ≤ i ≤ 6 • i + 1;
        println(res);
        res := MAXi|0 ≤ i ≤ 6 • i + 1;
        println(res);
        res := MINi|0 ≤ i ≤ 6 • i + 1;
        println(res);
        if (false ⇒ 4/0 ≤ 0) then
            println (1);
        if (true ⇐ 4/0 ≤ 0) then
            println (2)
    end
  initialization
    begin
        x := 4;
        y := 5
    end
end
```

Assertion checking cab be turned on and off. We can first test the usage of quantifiers and the handling of undefinedness in implication and consequence expressions by turning off the assertion checking. After we turn on assertion checking, we test the reorganizations of assertion clauses. Lime should check the class invariant after the *initialization* and every method call, check precondition, postcondition before, after method execution, respectively. After a violation of assertion is detected and reported, we use a negation ¬ to correct it and continue the test. The test result is attached in Appendix B. The test results show Lime handles assertions and undefinedness properly.

5.3 A Small System

We use a small system shown in Figure 5.2 to demonstrate the usage of the new Lime features. We use it to test specification inheritance and the notation *old* and *result*. This system is also used for demonstrating documentation generation. The source code for this system and test result are attached in Appendix C.

In this system, class *Add* contains a method *sum1to9* that is implemented using a *while* loop. For testing purpose, we put quantifier $\forall$ and notation *old* and *result* in its postcondition:

$$post\ (\forall\ i \mid 0 < i < 10\ \bullet\ i < result) \wedge (old(sum) < result)$$

Abstract class *AddSpe* gives a new specification for method *sum1to9*

$$post\ result = \Sigma\ i \mid 0 < i < 10\ \bullet\ i$$

Class *AA* is a subclass of *Add* and a subtype of *Add* and *AddSpe*. It contains a method *ave1to9* used for calculating the average of sequence 1, 2, ..., 9. We use an existential quantifier in its postcondition

$$post\ (\exists\ i \mid 1 \leq i \leq 9\ \bullet\ i < result) \Rightarrow (\exists\ i \mid 1 \leq i \leq 9\ \bullet\ i > result)$$

Class *AA* inherits the implementation of method *sum1to9* from class *Add*. It overrides it with a different implementation. Since it is a subtype of *Add* and *AddSpe*, it has to preserve the behavior specified in *Add* and *AddSpe*. The test result shows all the quantifiers, notations, and extended expression work properly. Here, we use a different way to test what kinds of assertions are checked in run-time; we take the fragment of the generated intermediate file *AA.lime.LIME* to show the reorganized assertions. This generating process has been explained in the Section 4.4.

```
public method sum1to9: integer

    begin
```

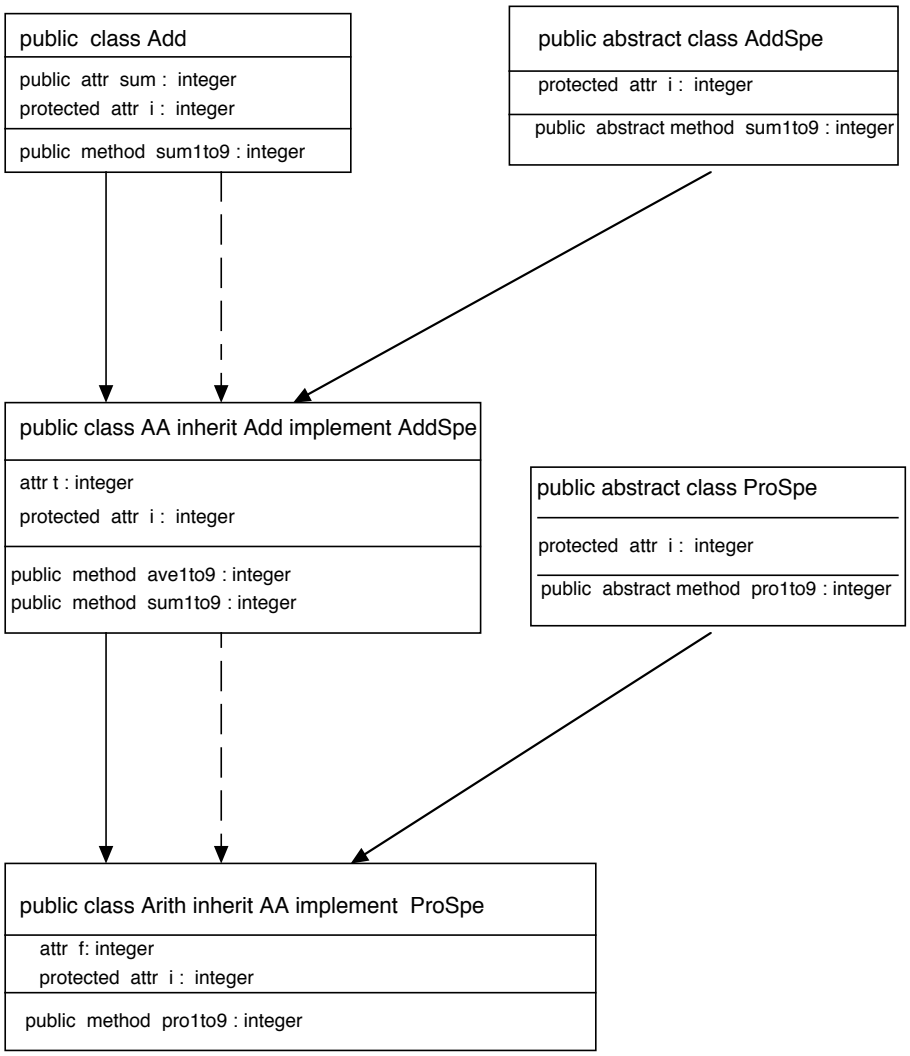

Figure 5.2: A Small System

monitorenter;

notifyAll;

/* following is the actual body of the method */

pre (sum = 0);

****

precondition *pre (sum = 0)* is from *Add*

****

old0 := old(sum);

****

evaluate expression in *old* notation and assign to a special variable

****

begin

sum := (1 + 9)*9/2;

println(sum);

return sum

end;

checkInvariant (sum 0) ;

****

this invariant is from *Add*

****

$post(result = \Sigma\ i \mid 0 < i < 10 \bullet\ i) \wedge ((\forall\ i \mid 0 < i < 10 \bullet\ i < result) \wedge (old0 < result))$;

****

in this postcondition, $(result = \Sigma\ i \mid 0 < i < 10 \bullet\ i)$ is from *AddSpe*

$(\forall\ i \mid 0 < i < 10 \bullet\ i < result) \wedge (old0 < result)$ is from *Add*

and the *old* notation has been replaced by its value at pre-state.

```
        **************************************************
        monitorexit
    end
```

The above generated intermediate code shows how subtype preserves the behavior of its supertypes. That is the intention of the design.

5.4 Generated Documents

We use our LimeD to generate documentations for the system shown in Figure 5.2. For a project, LimeD generates a class links and a summary page that uses a table to hold all class declarations in which the class name is a link to its corresponding document. The class links and summary page are located in the left and the right frames respectively shown in Figure 5.3.

LimeD generates documentation for each class. Class *Arith*'s document is shown in Figures 5.4, 5.5, and 5.6. The class documentation contains *Class Invariant*, *Class Hierarchy*, *Type Hierarchy*, *Attribute* and *Method* parts and their links are on the top and bottom of the page.

The *Class Invariant* part contains invariants defined in current class and inherited from supertypes. It also gives the invariant that should be preserved by the current class. It is shown in Figure 5.4.

The *Class Hierarchy* and *Type Hierarchy* parts show the inheritance relationship. The *Attribute* part represents the attributes declared in current class. The inherited attributes are given in the *Inherited Attribute* part. These are shown in Figure 5.5.

The *Method* part contains the method signature and specification; that is, the preconditions and postconditions defined in current class and inherited from super-

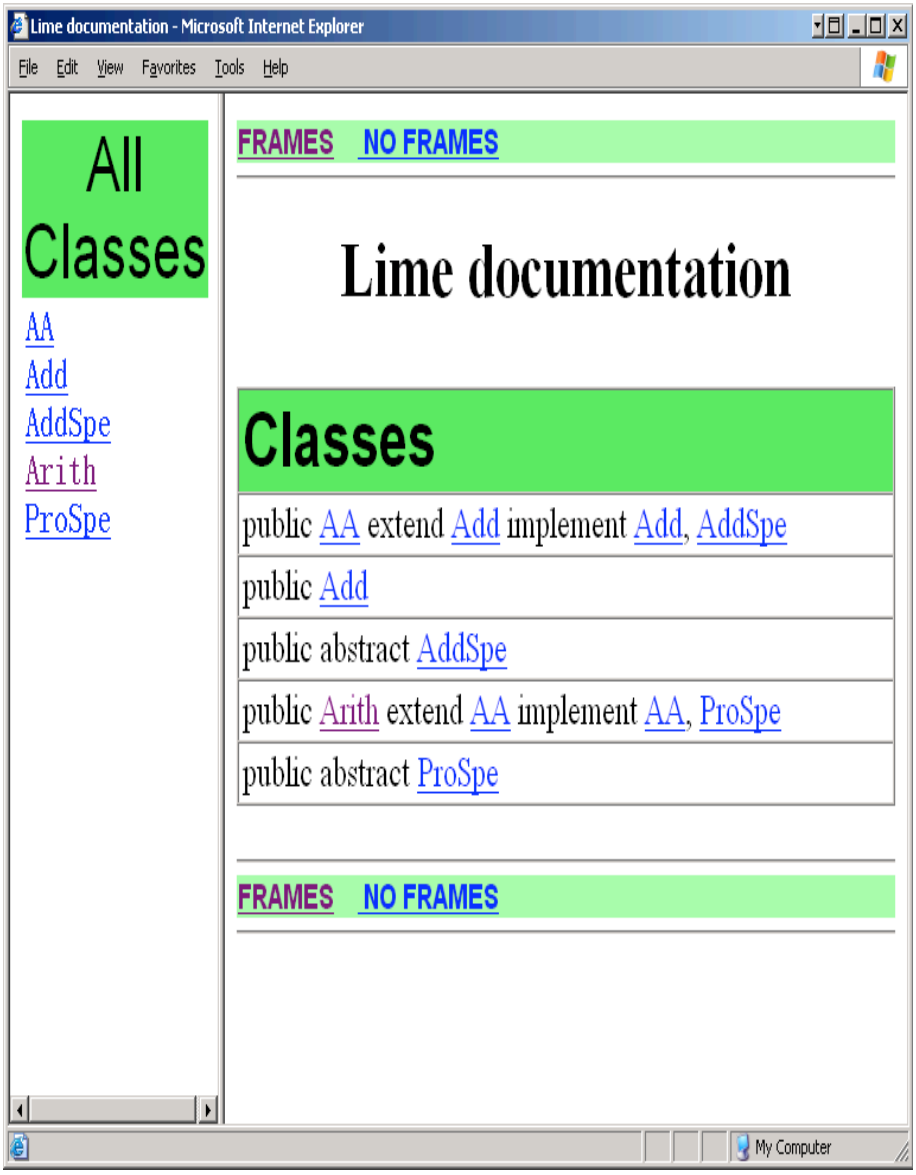

Figure 5.3: Summary Page

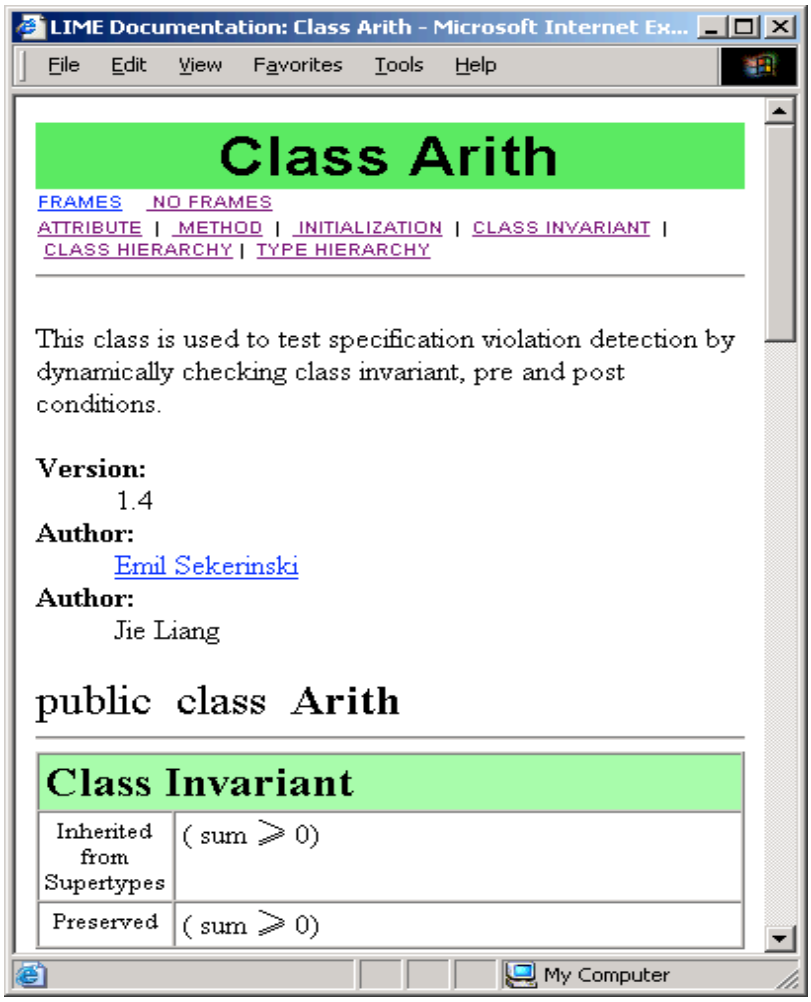

Figure 5.4: Class Arith Document Part One

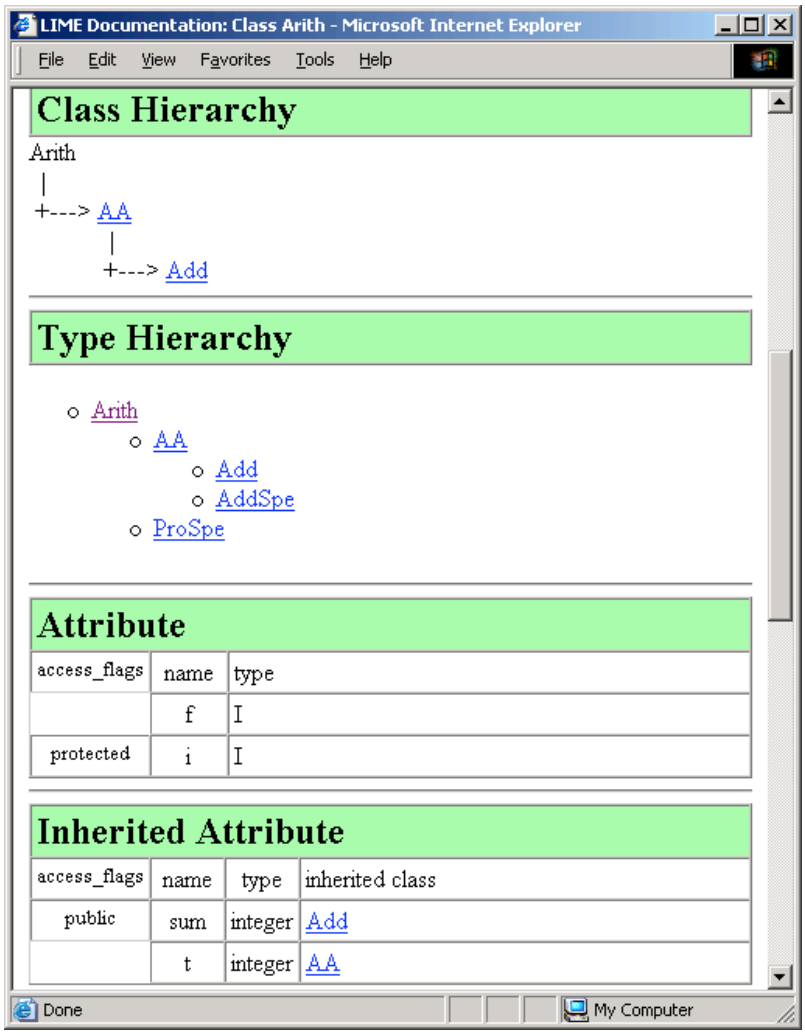

Figure 5.5: Class Arith Document Part Two

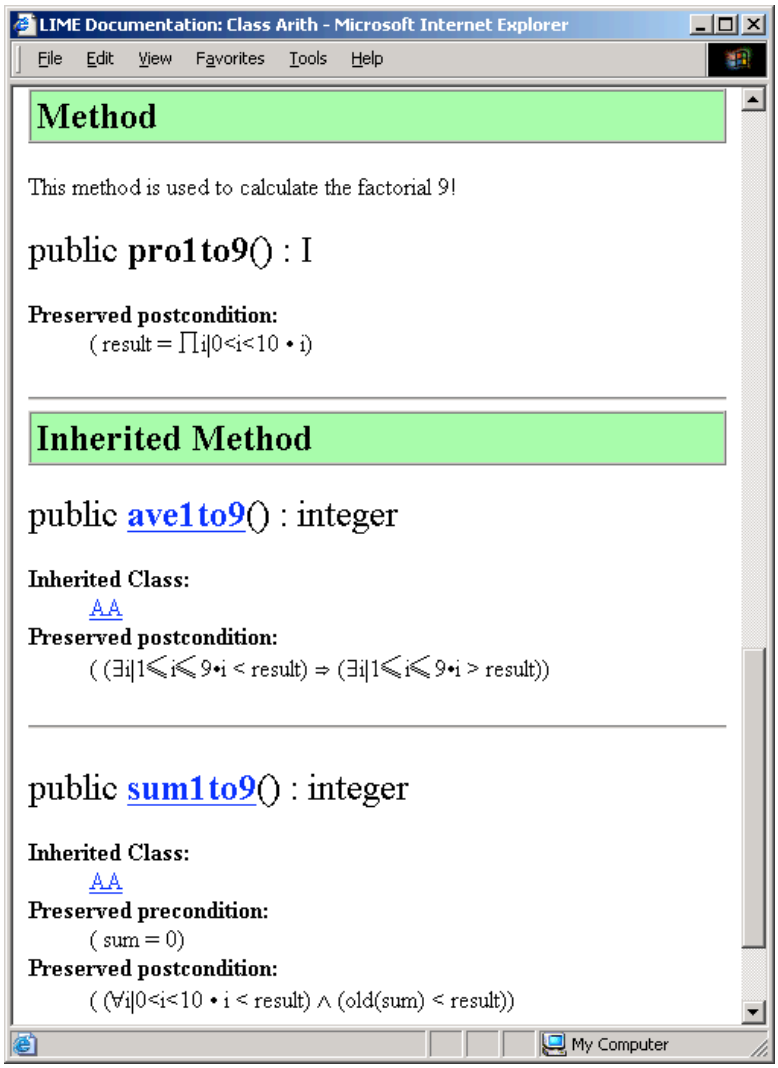

Figure 5.6: Class Arith Document Part Three

types. It also gives the preserved preconditions and postconditions. The inherited methods are in the *Inherited Method* part. They are shown in Figure 5.6.

Chapter 6

Conclusions

6.1 Summary

In this thesis, we present the development of the following Lime language features:

- Separating subclassing from subtyping. An inheritance mechanism has been proposed to separate subclassing from subtyping, to make the inheritance in Lime more flexible.

- Integrating specification into the language. The preconditions, postconditions and invariant clauses are used in Lime for expressing the behavioral specification that can be checked at run-time.

- Supporting automatic documentation generation. A documentation tool LimeD has been developed for generating behavioral interface documentation.

- Providing high expressiveness. Lime expression can be written using quantifiers and other standard mathematical notations.

6.2 Conclusion

Based on the above summarization, we conclude:

1. The inheritance mechanism proposed for Lime supports separating subclassing from subtyping so that code reuse can remove the restriction of satisfying the requirement of subtyping. It also allows the developing class selectively inherits the required contents from the existing classes. This mechanism can help in improving software reusability.

2. The integrated behavioral specification in Lime can help improving software correctness.

3. The documentation tool LimeD generates up-to-date documents for the existing system that is very helpful for improving software maintainability and extendibility.

4. The usage of quantifiers and standard mathematical notation can help improving software readability.

6.3 Future Work

One possible direction to extend our work is to combine assertions with concurrency. It would be a challenging research direction to provide an exception mechanism that can handle assertions and concurrency.

Appendix A

Java Source Code

Method *handleInheritance* in Class *JavaClassHandler*:

```
/** static Method handleInheritance puts all inherited MethodInfo to method table
  * by Depth First Search order First parser super class and interfaces file and
  * generate JavaClass, get all inherited methods information from JavaClass.
  * use the obtained method information to generte MethodInfos and put them to
  * method table for later static checking.
  */

  public static void handleInheritance(String supFilePath, boolean isInterface,
                                       boolean isSupertype, boolean beAbstract, boolean
                                       isInherit, MethodTable mT, int inheritLevel) {

      JavaClass supClass = null;
      File supFile;

      // isInterface used for creating File
      if (isInterface)
          supFile = new File(supFilePath + "_C.class");
```

```
else
    supFile = new File(supFilePath + ".class");

if (!supFile.exists()) {
    System.out.println("Error: super Class or Type file "
                        + supFile.toString() + " does not exist.");
} else {
    try {
        supClass = new ClassParser(supFile.getAbsolutePath()).parse();
    }
    catch (IOException ioe) {
        System.out.println(ioe);
    }
    catch (ClassFormatError cfe) {
        System.out.println(cfe);
    }

    String superName = supClass.getSuperclassName();
    String [] interfaceNames = supClass.getInterfaceNames();

    boolean inherit = false;    // "inherit" and "isInherit" used for
                                // avoid repeatly checking assertion

    for (int i = 0; i < interfaceNames.length; i++) {
        if (!supClass.getClassName().equals(interfaceNames[i] + "_C")) {
            if (superName.equals(interfaceNames[i] + "_C")) {
                inherit = true;
                handleInheritance(interfaceNames[i], true, isSupertype, true,
                                  true, mT, inheritLevel + 1);
            } else
```

```
                    handleInheritance(interfaceNames[i], true, isSupertype, true,
                                                isInherit, mT, inheritLevel + 1);
            }
        }

        if (!(superName.equals("java.lang.Object"))) {
            handleInheritance(superName, false, false, beAbstract,
                                    isInherit || inherit, mT, inheritLevel + 1);
        }

        MethodInfo [] methodInfos
         = new MethodInfoGenerator(isInterface, isSupertype, beAbstract,
               isInherit).createMethodInfos(supClass, inheritLevel);
        for (int i = 0; i < methodInfos.length; i++) {
            String mName = methodInfos[i].getMethodName();
            if (!( mName.equals("<init>") || methodInfos[i].isPrivate()
                || (mName.startsWith("check") && mName.endsWith("Assertion")))) {
                mT.addMethodInfo(methodInfos[i]);
            }
        }
    }
}
```

Method *addMethodInfo* in Class *MethodInfo*:

```
/**
 *  Method addMethodInfo add MethodInfo to method table and check assign weaker
 *  access privileges error collect all the method assertion info. from super types
 * for Behavioural subtyping,
 */
```

```
public void addMethodInfo (MethodInfo mI) {
    // use method name and argument as hask table key
    String key = mI.getMethodName() + mI.getMethodArgument();

    if(!mTable.containsKey(key)) { //case: method does not exist in method table
        mTable.put(key, mI);
        if (mI.isAbstract()) {
            numAbsMethod++;
        }
    } else {                        //case: method already in method table
        MethodInfo tempMI = (MethodInfo) mTable.get(key);
        if(JavaClassHandler.isAncestor(mI.getClassName(), tempMI.getClassName())) {
            //case: the existing method in ancestor class, check weaker access privilege
            //      and overrider it in method table

            //check access privilege
           try {
                checkAccessPrivilege(mI, tempMI);
            } catch (WeakerAccessException e) {
            }

            // merge method assertions
            String hMethodPre = tempMI.getMethodPre();
            hMethodPre = hMethodPre.replaceAll("\"", "");
            String lMethodPre = mI.getMethodPre();
            lMethodPre = lMethodPre.replaceAll("\"", "");
            if (hMethodPre.length() > 0) {
                if (lMethodPre.length() > 0)
```

mI.setMethodPre(lMethodPre + “$\bigvee$” + hMethodPre);

```
            else
                mI.setMethodPre(hMethodPre);
        }

        String hMethodPost = tempMI.getMethodPost();
        hMethodPost = hMethodPost.replaceAll("\"", "");
        String lMethodPost = mI.getMethodPost();
        lMethodPost = lMethodPost.replaceAll("\"", "");
        if (hMethodPost.length() > 0) {
            if (lMethodPost.length() > 0)
                mI.setMethodPre(lMethodPre + "⋀" + hMethodPre);
            else
                mI.setMethodPost(hMethodPost);
        }

        mTable.remove(key);
        mTable.put(key, mI);
        if(mI.isAbstract() && !tempMI.isAbstract()) {
            numAbsMethod++;
        }
        if(!mI.isAbstract() && tempMI.isAbstract()) {
            numAbsMethod--;
        }

    } else {
        // two methods in differen branch in inherited tree, only thing need
        // to do override abstract method by non abstract one

        // for Behavioural subtype, the method Info in method table should
```

```
// hold all assertion info.
if(!mI.isAbstract() && tempMI.isAbstract()) {

    String hMethodPre = tempMI.getMethodPre();
    hMethodPre = hMethodPre.replaceAll("\"", "");
    String lMethodPre = mI.getMethodPre();
    lMethodPre = lMethodPre.replaceAll("\"", "");
    if (hMethodPre.length() > 0) {
        if (lMethodPre.length() > 0)

            mI.setMethodPre(lMethodPre + "⋁" + hMethodPre);

        else
            mI.setMethodPre(hMethodPre);
    }

    String hMethodPost = tempMI.getMethodPost();
    hMethodPost = hMethodPost.replaceAll("\"", "");
    String lMethodPost = mI.getMethodPost();
    lMethodPost = lMethodPost.replaceAll("\"", "");
    if (hMethodPost.length() > 0) {
        if (lMethodPost.length() > 0)

            mI.setMethodPost(lMethodPost + "⋀" + hMethodPost);

        else
            mI.setMethodPost(hMethodPost);
    }

    mTable.remove(key);
    mTable.put(key, mI);
    numAbsMethod--;
```

```
            } else {
                String hMethodPre = tempMI.getMethodPre();
                hMethodPre = hMethodPre.replaceAll("\"", "");
                String lMethodPre = mI.getMethodPre();
                lMethodPre = lMethodPre.replaceAll("\"", "");
                if (lMethodPre.length() > 0) {
                    if (hMethodPre.length() > 0)

                        tempMI.setMethodPre(hMethodPre + "⋁" + lMethodPre);

                    else
                        tempMI.setMethodPre(lMethodPre);
                }

                String hMethodPost = tempMI.getMethodPost();
                hMethodPost = hMethodPost.replaceAll("\"", "");
                String lMethodPost = mI.getMethodPost();
                lMethodPost = lMethodPost.replaceAll("\"", "");
                if (lMethodPost.length() > 0) {
                    if (hMethodPost.length() > 0)

                        tempMI.setMethodPost(hMethodPost + "⋀" + lMethodPost);

                    else
                        tempMI.setMethodPost(lMethodPost);
                }
            }
        }
    }
}
```

Appendix B

Fundamental and Assertion Test Results

B.1 Lime Source Code

```
abstract class A
  attr x: integer;
  const z = 4;
  public method m1
    begin
     // z := 2;
      x := 10;
      println(x)
    end
    protected abstract method m2 : integer
end  // end of A


class B inherit A
```

```
  public method m2 :integer
    begin
      return 2
    end
end  // end of B

// class C implement B
abstract class C implement B
    public method m1
    begin
        println(z)
    end
 end  // end of C

class D extend B
  public method m3
    begin
        x := 3;
        println(x)
    end
end    // end of D

class E extend B implement C
  attr y : integer;
  //private method m1
  public method m1
    begin
        y := 14;
        println(y)
    end
```

```
end   // end of E


program T
     var a : A
     var c : C
     var d : D
  begin
    //obj a := new A();
    obj a := new B();
    call a.m1;
    obj a := new E();
    call a.m1;
    obj d := new D()
    //a := d
  end   // end of T
```

B.2 Fundamental Test Results

The test results has been simplified and some explanations are added:

```
[stan:~] liangj2% cd work/LimeC
[stan:~/work/LimeC] liangj2% java LimeC -IP A.lime
Lime Parser:  Reading from file A.lime . . .
Error: can not to assign a value to constant variable z
Lime Parser:  Encountered errors during parse.
************************************************************************
detect assigning value to constant variable
we comment out this assignment statement and compile again
************************************************************************
[stan:~/work/LimeC] liangj2% java LimeC -IP A.lime
```

```
Lime Parser:  Reading from file A.lime . . .
Lime Parser:  Lime program parsed successfully.
[stan:~/work/LimeC] liangj2% java LimeC -J A.lime.LIME
Lime Parser:  Reading from file A.lime.LIME . . .
Lime Parser:  Lime program parsed successfully.
[stan:~/work/LimeC] liangj2% jasmin A.j
Generated: A.class
[stan:~/work/LimeC] liangj2% jasmin A_C.j
Generated: A_C.class
[stan:~/work/LimeC] liangj2% java LimeC -IP B.lime
......
[stan:~/work/LimeC] liangj2% java LimeC -IP C.lime
Lime Parser:  Reading from file C.lime . . .
Error:  C should be declared abstract; it does not define the following methods
m2 () : integer in class B
**************************************************************************
detect declaring an non abstract class with non implemented method
we change its declaration to abstract class and recompile.
**************************************************************************
......
[stan:~/work/LimeC] liangj2% java LimeC -IP D.lime
......
[stan:~/work/LimeC] liangj2% java LimeC -IP E.lime
Lime Parser:  Reading from file E.lime . . .
Error: m1 () in E can not override m1 () in A; attempting to assign weaker access privileges.
**************************************************************************
detect assigning weaker access privilege
we change m1's access modifier from private to public and recompile.
**************************************************************************
[stan:~/work/LimeC] liangj2% java LimeC -IP E.lime
```

```
......
[stan:~/work/LimeC] liangj2% java LimeC -IP T.lime
Lime Parser:  Reading from file T.lime . . .
Error: A is abstract; can not be instanced.
Error: incompatible types; d's type (D) is not the subtype of a's type (A)
a :=  d
Lime Parser:  Encountered errors during parse.
**************************************************************************
detect creating an instance from an abstract class
detect incompatible type assignment
we comment out both of lines and recompile.
**************************************************************************
[stan:~/work/LimeC] liangj2% java LimeC -IP T.lime
......
[stan:~/work/LimeC] liangj2% java T
10
14
**************************************************************************
These two lines show polymorphism in Lime
The first is from the execution of method in class B
The second is from the execution of method in class E
**************************************************************************
[stan:~/work/LimeC] liangj2%
```

B.3 Assertion Test Results

```
[stan:~/work/LimeC] liangj2% java LimeC -IP AS.lime
Lime Parser:  Reading from file AS.lime . . .
Lime Parser:  Lime program parsed successfully.
```

```
[stan:~/work/LimeC] liangj2% java LimeC -J AS.lime.LIME
************************************************************************
We use -J to turn asseriton checking off
************************************************************************
Lime Parser:  Reading from file AS.lime.LIME . . .
Lime Parser:  Lime program parsed successfully.
[stan:~/work/LimeC] liangj2% jasmin AS.j
Generated: AS.class
[stan:~/work/LimeC] liangj2% jasmin AS_C.j
Generated: AS_C.class
[stan:~/work/LimeC] liangj2% java LimeC -IP TA.lime
Lime Parser:  Reading from file TA.lime . . .
......
[stan:~/work/LimeC] liangj2% java TA
28
************************************************************************
This is the sum of sequence 1, 2 ... 7
************************************************************************
5040
************************************************************************
This is the product of sequence 1, 2 ... 7 that is 7!
************************************************************************
7
************************************************************************
This is the maximum of sequence 1, 2 ... 7
************************************************************************
1
************************************************************************
This is the minimum of sequence 1, 2 ... 7
************************************************************************
```

```
1
**************************************************************************
```

When implication false $\Rightarrow$ 4/0 evaluates true, print it out.

undefined 4/0 is handled.

```
**************************************************************************
2
**************************************************************************
```

print out for true $\Leftarrow$ 4/0.

```
**************************************************************************
```

We turn assertion checking on

```
[stan:~/work/LimeC] liangj2% java LimeC -IP AS.lime
Lime Parser:  Reading from file AS.lime . . .
Lime Parser:  Lime program parsed successfully.
[stan:~/work/LimeC] liangj2% java LimeC -JA AS.lime.LIME
**************************************************************************
We use -JA to turn asseriton checking off
**************************************************************************
......
[stan:~/work/LimeC] liangj2% java TA
Class invariant assertion failed in method Initialization of class AS
abnormal program termination
[stan:~/work/LimeC] liangj2%
**************************************************************************
initialization x:= 4; makes class invariant x = 3 checking fail.
we negate invariant, continue test.
**************************************************************************
```

```
[stan:~/work/LimeC] liangj2% java TA
Precondition assertion failed in method dbcTest of class AS
abnormal program termination
**************************************************************************
precondition x = y fails since x := 4; y := 5 in initialization.
negate precondition and continue test.
**************************************************************************
[stan:~/work/LimeC] liangj2% java TA
Assertion failed in method dbcTest of class AS
abnormal program termination
**************************************************************************
assert (x > y ) fails, negate it, continue test.
**************************************************************************
[stan:~/work/LimeC] liangj2% java TA
28
5040
7
1
1
2
Class invariant assertion failed in method dbcTest of class AS
abnormal program termination
**************************************************************************
class invariant fails again, since x := 3; inside method.
we comment out invariant. continue test.
**************************************************************************
[stan:~/work/LimeC] liangj2% java TA
28
5040
7
```

```
1
1
2
Postcondition assertion failed in method dbcTest of class AS
abnormal program termination
**************************************************************************
postcondition y = res fails since y := 5 in initialization, res holds 1.
negate precondition and continue test.
**************************************************************************
[stan:~/work/LimeC] liangj2% java TA
28
5040
7
1
1
2
**************************************************************************
Right now, everything is fine
**************************************************************************
```

Appendix C

A Small System

C.1 Source Code

```
{ @description This class is used to test specification violation detection by
        dynamically check class invariant, pre and post condition.
  @author <A href ="http://www.cas.mcmaster.ca/~emil">Emil Skerinski</A>
  @author Jie Liang
  @version 1.4
}
public class Add
  public attr sum : integer;
  protected attr i : integer;
```

invariant $\text{sum} \geq 0$

```
 {@description this method used for calculating sum of sequence 1,2,...,9 }
  public method sum1to9: integer
    pre sum = 0
```

post $(\forall\ i \mid 0 < i < 10 \bullet i < result) \wedge (old(sum) < result)$

```
    begin
      i := 1;
      while i < 10 do
        begin
          sum := sum + i;
          i := i + 1
        end;
      return sum
    end
   initialization
     sum := 0
end  // end of Add

{ @description This class gives a specification of adding the sequence. }
public abstract class AddSpe
    protected attr i : integer;
    public abstract method sum1to9: integer
```

$$\text{post } result = \Sigma\ i \mid 0 < i < 10 \bullet i$$

```
end  \\ end of AddSpe

{ @description This class is used to test specification violation detection by
        dynamically check class invariant, pre and post condition.
  @author <A href ="http://www.cas.mcmaster.ca/~emil">Emil Skerinski</A>
  @author Jie Liang
  @version 1.4
}
public class AA inherit Add implement AddSpe
    attr t: integer;
```

```
    protected attr i: integer;
 {@description This method is used to calculate the average of sequence of 1..9 }
   public method ave1to9 : integer
```

$$\text{post } (\exists\, i \mid 1 \leq i \leq 9 \bullet i < result) \Rightarrow (\exists i \mid 1 \leq i \leq 9 \bullet i > result)$$

```
      begin
         t := call sum1to9;
         t := t/9;
         println(t);
         return t
      end
{@description This method overrides the inherited one with different implementation }
  public method sum1to9: integer
    begin
      sum := (1 + 9)*9/2;
      println(sum);
      return sum
    end
end  // end of AA

{ @description This class gives a specification of production the sequence 1, 2, ...9}
public abstract class ProSpe
    protected attr i : integer;
    public abstract method pro1to9: integer
```

$$\text{post } result = \Pi i \mid 0 < i < 10 \bullet i$$

```
end  //end of ProSpe

{ @description This class is used to test specification violation detection by
```

```
       dynamically check class invariant, pre and post condition.
  @author <A href ="http://www.cas.mcmaster.ca/~emil">Emil Skerinski</A>
  @author Jie Liang
  @version 1.4
}
public class Arith inherit AA implement ProSpe
  attr f: integer;
  protected attr i: integer;
{@description This method is used to calculate the factorial 9!}
  public method pro1to9 : integer
    begin
      f := 1;
      i := 1;
        while i  9 do
          begin
            f := f * i;
            i := i + 1
          end;
        println(f);
      return f
    end
end  // end of Arith

program TA
    var ar : Arith
    var aa : AA
  begin
    obj ar := new Arith();
    call ar.ave1to9;
    call ar.sum1to9;
```

```
    call ar.pro1to9
  end
```

C.2 Test Results

```
[stan:~/work/LimeC] liangj2% java LimeC -IP Add.lime
Lime Parser:  Reading from file Add.lime . . .
Lime Parser:  Lime program parsed successfully.
[stan:~/work/LimeC] liangj2% java LimeC -JA Add.lime.LIME
Lime Parser:  Reading from file Add.lime.LIME . . .
Lime Parser:  Lime program parsed successfully.
[stan:~/work/LimeC] liangj2% jasmin Add.j
Generated: Add.class
[stan:~/work/LimeC] liangj2% jasmin Add_C.j
Generated: Add_C.class
[stan:~/work/LimeC] liangj2% java LimeC -IP AddSpe.lime
......
[stan:~/work/LimeC] liangj2% java LimeC -IP AA.lime
......
[stan:~/work/LimeC] liangj2% java LimeC -IP ProSpe.lime
......
[stan:~/work/LimeC] liangj2% java LimeC -IP Arith.lime
......
[stan:~/work/LimeC] liangj2% java LimeC -IP TA.lime
Lime Parser:  Reading from file TA.lime . . .
Lime Parser:  Lime program parsed successfully.
[stan:~/work/LimeC] liangj2% java LimeC -JA TA.lime.LIME
Lime Parser:  Reading from file TA.lime.LIME . . .
Lime Parser:  Lime program parsed successfully.
```

```
[stan:~/work/LimeC] liangj2% jasmin TA_C.j
Generated: TA.class
[stan:~/work/LimeC] liangj2% java TA
45
*********************************************************************************
method ave1to9 uses sum1to9, printed from sum1to9
*********************************************************************************
5
*********************************************************************************
the result of ave1to9
*********************************************************************************
45
*********************************************************************************
the result of sum1to9
*********************************************************************************
362880
*********************************************************************************
the result of pro1to9
*********************************************************************************
```

Bibliography

[1] G. Adams, "Internationalization and character set standards," *StandardView*, vol. 1, no. 1, pp. 31–39, 1993.

[2] P. America, "Designing an object-oriented programming language with behavioural subtyping," in *Foundations of Object-Oriented Languages, Lecture Notes in Computer Science* (J. de Bakker, W. de Roever, and G. Rozenberg, eds.), vol. 489, (REX School/Workshop, Noordwijkerhout, The Netherlands, May/June 1990), pp. 60–90, Springer-Verlag, 1991.

[3] P. America and F. van der Linden, "A parallel object-oriented language with inheritance and subtyping," in *Proceedings OOPSLA/ECOOP '90, ACM SIGPLAN Notices*, vol. 25, pp. 161–168, October 1990.

[4] A. W. Appel, *Modern Compiler Implementation in Java*. Cambridge University Press, 1998.

[5] A. Avenarius and S. Oppermann, "FWEB: a literate programming system for fortran8x," *ACM SIGPLAN Notices*, vol. 25, pp. 52–58, January 1990.

[6] D. Bartetzko, C. Fischer, M. Moller, and H. Wehrheim, "Jass - Java with assertions," in *Proceedings of the First Workshop on Runtime Verification, Electronic*

Notes in Theoretical Computer Science (K. Havelund and G. Rosu, eds.), vol. 55, Elsevier Science, July 2001.

[7] G. M. Birtwistle, O. J. Dahl, B. Myhrhaug, and K. Nygaard, *SIMULA Begin*. Auerbach, 1973.

[8] K. B. Bruce, A. Schuett, and R. van Gent, "PolyTOIL: A type-safe polymorphic object-oriented language," *ACM TOPLAS*, vol. 25, no. 2, pp. 225–290, March 2003.

[9] T. Budd, *An Introduction to Object-Oriented Programming*. Addison Wesley, third ed., 2002.

[10] L. Burdy, Y. Cheon, D. Cok, M. D. Ernst, J. Kiniry, G. T. Leavens, K. R. M. Leino, and E. Poll, "An overview of JML tools and applications," in *Eighth International Workshop on Formal Methods for Industrial Critical Systems (FMICS 03), Electronic Notes in Theoretical Computer Science*, vol. 66, (Trondheim, Norway), pp. 1–17, Elsevier Science, June 5–7, 2003.

[11] C. Chen, "A Tool for Detecting Fragile Base Class Problem in Java," Master's thesis, Department of Computing and Software, McMaster University, 2002.

[12] Y. Cheon and G. T. Leavens, "A Runtime Assertion Checker for the Java Modeling Language (JML)," in *International Conference on Software Engineering Research and Practice (SERP)*, pp. 322–328, Las Vegas, Nevada, USA: Computer Science Research, Education, and Applications (CSREA) Press, June 2002.

[13] W. R. Cook, W. L. Hill, and P. S. Canning, "Inheritance is not subtyping," in *Proceedings of the ACM Conference on Principles of Programming Languages*

(POPL '90), (San Francisco), pp. 125–135, ACM Press. Addison-Wesley, January 1990.

[14] F. Cristian, "Exception handling and software fault tolerance," *IEEE Transactions on Computers*, vol. 31, no. 6, pp. 531–540, June 1982.

[15] E. Cusack, "Inheritance in object oriented Z," in *ECOOP '91 European Conference on Object-Oriented Programming, Lecture Notes in Computer Science* (P. America, ed.), vol. 512, (Geneva, Switzerland), pp. 167–179, Springer-Verlag, July 1991.

[16] M. Day, R. Gruber, B. Liskov, and A. C. Myers, "Subtypes vs. where clause: Constraining parametric polymorphism," in *Proceedings of the tenth annual conference on Object-Oriented Programming systems, languages and applications, ACM SIGPLAN Notices*, vol. 30, (Austin, TX, USA), pp. 156–168, October 1995.

[17] K. K. Dhara and G. T. Leavens, "Forcing behavioral subtyping through specification inheritance," in *Proceedings of the 18th international Conference on Software Engineering,*, (Berlin, Germany), pp. 258–267, IEEE Computer Society Press, March 1996.

[18] M. J. Dürst, "Uniprep - preparing a C/C++ compiler for Unicode," *SIGPLAN Not.*, vol. 29, no. 1, p. 53, 1994.

[19] H. Eriksson, E. Berglund, and P. Nevalainen, "Using knowledge engineering support for a Java documentation viewer," in *Proceedings of the 14th international conference on Software engineering and knowledge engineering*, pp. 57–64, ACM Press, 2002.

[20] L. Friendly, "The design of distributed hyperlinked programming documentation," in *Proceedings of the 1995 International Workshop on Hypermedia Design, (IWHD'95)* (S. Frasse, F. Garsotto, T. Isakowitz, J. Nanard, and M. Nanard, eds.), (Montpellier, France), pp. 151–173, Springer, June 1995.

[21] J. B. Goodenough, "Exception handling: Issues and a proposed notation," *Communications of the ACM*, vol. 18, no. 12, pp. 683–696, 1975.

[22] J. Gosling, B. Joy, G. Steele, and G. Bracha, *The Java Language Specification Second Edition*. Addision-Wesley, 2000.

[23] C. A. R. Hoare, "An axiomatic basis for computer programming," *Communications of the ACM*, vol. 12, no. 10, pp. 576–583, 1969.

[24] A. Johnson and B. Johnson, "Literate programming using noweb," *Linux Journal*, vol. 1997, issue 42, October 1997.

[25] C. B. Jones, *Systematic Software Development Using VDM*. Englewood Cliffs, N.J.: Prentice-Hall International Series in Computer Science, second ed., 1990.

[26] S. Khurshid, D. Marinov, and D. Jackson, "An analyzable annotation language," in *ACM SIGPLAN Notices , Proceedings of the 17th ACM SIGPLAN conference on Object-Oriented Programming, Systems, Languages, and Applications*, vol. 37, (Seattle, Washington, USA), pp. 231–245, November 2002.

[27] D. E. Knuth, "Literate programming," *The Computer Journal*, vol. 27, no. 2, pp. 97–111, May 1984.

[28] D. E. Knuth and S. Levy, *The CWEB System of Structured Documentation*. Addision-Wesley, 1993.

[29] D. Kramer, "API documentation from source code comments: A case study of javadoc," in *Proceedings of the17th Annual International Conference on Computer Documentation*, (New Orleans, Louisiana, United States), pp. 147–153, ACM Press, September 1999.

[30] R. Kramer, "iContract - the Java design by contract tool," *TOOLS 26: Technology of Object-Oriented Languages and Systems*, pp. 295–307, 1998.

[31] B. B. Kristensen, O. L. Madsen, B. Moeller-Pedersen, and K. Nygaard, "The BETA programming language," in *Research Directions in Object-Oriented Programming* (B. D. Shriver and P. Wegner, eds.), MIT Press, 1987.

[32] W. R. LaLonde and J. Pugh, "Subclassing $\neq$ subtyping $\neq$ is-a," *Journal of Object-Oriented Programming*, vol. 3, no. 5, pp. 57–62, January 1991.

[33] L. Lamport, "A simple approach to specifying concurrent systems," *Communications of the ACM*, vol. 32, no. 1, pp. 32–45, 1989.

[34] G. T. Leavens, A. L. Baker, and C. Ruby, "JML: A notation for detailed design," in *Behavioral Specifications for Businesses and Systems, chapter 12 (The Kluwer International Series in Engineering and Computer Science)* (H. Kilov, B. Rumpe, and I. Simmonds, eds.), vol. 523, pp. 175–188, Kluwer Academic Publishers, 1999.

[35] D. M. Leslie, "Using javadoc and XML to produce API reference documentation," in *Proceedings of the 20th Annual International Conference on Computer Documentation*, (Toronto, Ontario, Canada), October 2002.

[36] J. Liberty, *Programming C#*. O'REILLY, 2001.

[37] T. Lindholm and F. Yellin, *The Java Virtual Machine Specification*. Addision-Wesley, second ed., 1997.

[38] B. H. Liskov and A. Snyder, "Exception handling in CLU," *IEEE Transactions on Software Engineering*, vol. 5, no. 6, pp. 546–558, November 1979.

[39] B. H. Liskov and J. M. Wing, "A behavioral notion of subtyping," *ACM Transactions on Programming Languages and Systems*, vol. 16, no. 6, pp. 1811–1841, November 1994.

[40] B. H. Liskov and J. M. Wing, "Behavioral subtyping using invariants and constrains," *Technical Reports CMU CS-99-56, School of Computer Science, Carnegie Mellon University*, July 1999.

[41] B. H. Liskov and J. M. Wing, "Subtyping in state based approaches," in *Formal Methods for Distributed Processing, A survey of Object-Oriented Approaches* (H. Bowman and J. Derrick, eds.), Cambridge University Press, 2001.

[42] G. Lou, "A compiler for an action-based object-oriented programming language," Master's thesis, Department of Computing and Software, McMaster University, 2003. a M. Sc. Thesis, draft.

[43] S. Marlow, "Haddock, a Haskell document tool," in *Proceedings of the ACM SIGPLAN Workshop on Haskell*, (Pittsburgh, Pennsylvania), ACM Press, October 2002.

[44] B. Meyer, "Applying "design by contract"," *Computer*, vol. 25, no. 10, pp. 40–51, October 1992.

[45] B. Meyer, *Object-Oriented Software Construction 2nd edition*. Prentice-Hall, 1997.

[46] J. Meyer and T. Downing, *Java Virtual Machine*. O'Reilly, 1997.

[47] A. Mikhajlova and E. Sekerinski, "Class refinement and interface refinement in object-oriented development," in *Fourth International Formal Methods Europe Symposium, FME'97, Lecture Notes in Computer Science* (J. Fitzgerald, C. Jones, and P. Lucas, eds.), vol. 1313, (Graz, Austria), pp. 82–101, Springer-Verlag, September 1997.

[48] S. Murer, S. Omohundro, D. Stoutamire, and C. Szyperski, "Iteration abstraction in Sather," *ACM Transactions on Programming Languages and Systems*, vol. 18, no. 1, pp. 1–15, January 1996.

[49] D. L. Parnas, "A technique for software module specification with examples," *Communications of the ACM*, vol. 15, no. 5, pp. 330–336, 1972.

[50] R. Plösch, "Evaluation of assertion support for the Java programming language," *Journal of Object Technology*, vol. 1, no. 3 Special issue: TOOLS USA 2002 Proceedings, pp. 5–17, 2002.

[51] A. L. Powell, J. C. French, and J. C. Knight, "A systematic approach to creating and maintaining software documentation," in *Proceedings of the 1996 ACM symposium on Applied Computing*, (Philadelphia, Pennsylvania), pp. 201–208, February 1996.

[52] D. S. Rosenblum, "A practical approach to programming with asseritons," *IEEE Transactions on Software Engineering*, vol. SE-21, no. 1, pp. 19–31, January 1995.

[53] C. Schaffert, T. Cooper, B. Bullis, M. Kilian, and C. Wilpolt, "An introduction to Trellis/Owl," in *Proceedings OOPSLA '86, ACM SIGPLAN Notices*, pp. 9–16, September 1986.

[54] E. Sekerinski, "Concurrent object-oriented programs: From specification to code," in *First International Symposium on Formal Methods for Components and Objects, FMCO 02*, Lecture Notes in Computer Science 2852, (Leiden, The Netherlands), pp. 403–423, Springer-Verlag, 2003.

[55] B. Stroustrup, *C++*. Addision-Wesley, 1987.

[56] C. Szypersky, S. Omohundro, and S. Murer, "Engineering a programming language: The type and class system of Sather," in *Proceedings, First International Conference on Programming Languages and System Architectures, Lecture Notes in Computer Science* (J. Gutknecht, ed.), vol. 782, (Zurich, Switzerland), pp. 208–227, Springer Verlag, March 1994.

[57] J. M. Wing, "Writing Larch interface language specification," *ACM Transactions on Programming Languages and Systems*, vol. 9, no. 1, pp. 1–24, January 1987.

[58] J. Woodcock and J. Davies, *Using Z: Specification, Refinement, and Proof*. Prentice-Hall International Series in Computer Science, 1996.