Int J Software Informatics, Volume 10, Issue 4 (2016), pp. 000–000
International Journal of Software and Informatics, ISSN 1673-7288
©2016 by ISCAS. All rights reserved.

E-mail: ijsi@iscas.ac.cn
http://www.ijsi.org
Tel: +86-10-62661048

DOI: 10.21655/ijsi.1673-7288.00234

Formal Verification of 'Programming to Interfaces' Programs

Jianhua Zhao and Xuandong Li

(Department of Computer Science and Technology, Nanjing University, Nanjing 210023, P.R. China)

Abstract This paper presents a formal approach to specify and verify object-oriented programs written in the 'programming to interfaces' paradigm. In this approach, besides the methods to be invoked by its clients, an interface also declares a set of abstract and polymorphic function/predicate symbols, together with a set of constraints about these symbols. The methods declared in this interface are specified using these abstract symbols.

A class implementing this interface can give its own definitions to the abstract symbols, as long as all the constraints are satisfied. This class implements all the methods declared in the interface such that the method specification declared in the interface are satisfied w.r.t. the function symbol definitions in this class.

Based on the constraints about the abstract symbols, client code using the interfaces can be specified and verified precisely without knowing what classes implement the interfaces. Given more information about the implementing classes, the specifications of the client code can be specialized into more precise ones without re-verifying the client code.

Key words: code verification; object oriented; interface; polymorphism

Zhao JH, Li XD. Formal verification of 'programming to interfaces' programs. *Int J Software Informatics*, **Vol.10, No.4 (2016): 000–000. http://www.ijsi.org/1673-7288/10/234.htm**

1 Introduction

One of the important programming paradigms of OO programming is 'programming to interfaces'. Programmers can use an interface without knowing the details of its implementations. This programming paradigm decouples the code using the interfaces and the implementations of these interfaces. It also makes programs more flexible, because programmers can make a piece of client code using an interface fulfill different functional features by implementing the interface differently, without modifying the client code. So this paradigm is widely used, and supported by many modern OO programming languages.

There are already a number of research works on how to deal with inheritance and method overriding. To avoid re-verification of the client code using interfaces, researchers deal with method dynamic binding based on the Liskov Substitution Principle (LSP) sub-typing rule[1]. Roughly speaking, the overriding method should

Corresponding author: Jianhua Zhao, Email: zhaojh@nju.edu.cn
Received 2016-07-30; Revised 2016-12-12; Accepted 2017-01-07.

have a weaker precondition and a stronger post-condition. However, such approaches are not suitable for the 'programming to interfaces' paradigm because of two reasons.

– An interface declares no member variable. Those approaches can not specify methods precisely without referring to member variables.

– The behavioral sub-typing is too restrictive for the 'programming to interfaces' paradigm. In many cases, different implementations of an interface **should behave differently** such that the client code using the interface may have different functional features as expected.

Now we take the interface `java.lang.Comparable` in the standard package of Java as an example to show how 'programming-to-interface' paradigm should be treated. The interface `Comparable` has one method `CompareTo`, which compares an object with **this**. According to the Java documentation, a class implementing `Comparable` must ensure that the following formulas hold for any x, y, z.

$$\begin{aligned}
&\mathtt{sgn}(x{\rightarrow}\mathtt{compareTo}(y)) = -\mathtt{sgn}(y{\rightarrow}\mathtt{compareTo}(x));\\
&(x{\rightarrow}\mathtt{compareTo}(y) > 0 \wedge y{\rightarrow}\mathtt{compareTo}(z) > 0) \Rightarrow x{\rightarrow}\mathtt{compareTo}(z) > 0;\\
&(x{\rightarrow}\mathtt{compareTo}(y) = 0) \Rightarrow (\mathtt{sgn}(x{\rightarrow}\mathtt{compareTo}(z)) = \mathtt{sgn}(y{\rightarrow}\mathtt{compareTo}(z))).
\end{aligned}$$

Here `sgn` is a function which yields -1, 0 and $+1$ respectively when the parameter is less than, equal to or greater than 0. The above requirements mean that the method `compareTo` induces a total order over the objects.

The following code assigns the 'smaller' one of the objects $o1, o2$ to `s`.

$$\textbf{if } (\mathrm{o1}{\rightarrow}\mathtt{compareTo}(\mathrm{o2}){>}0)\ \ \mathtt{s} = \mathrm{o2};\ \ \textbf{else}\ \ \mathtt{s} = \mathrm{o1};$$

To support the 'programming to interfaces' paradigm, this piece of client code using `Comparable` must be specified and verified without referring to the classes implementing `Comparable`. This is critical because programmers should be able to add a new implementation of `Comparable` without having to re-verify the above client code. Further more, the specification of the client code must be precise and flexible enough, such that programmers can conclude that their new implementation makes the client code fulfill the functional features as expected. For example, suppose that `Comparable` is implemented by a class `Point` for points in the X-Y plate. The method `compareTo` in `Point` compares two points by their distances to the original point $(0, 0)$. Programmers should be able to conclude that if $o1$ and $o2$ refer to two `Point` objects, the above client code assigns `s` with the one closer to the original point without re-verifying the code.

In this paper, an approach is presented to specify and verify programs in the 'programming to interfaces' paradigm.

– **Abstract specifications of interface methods.** The interface methods are specified through a set of polymorphic function/predicate symbols and their constraints declared in the interface.

– **A flexible implementation relation between interfaces and classes.** To implement an interface, a class must define all the function symbols and implement all the methods declared in the interface. Two requirements are (1) the function symbol definitions must satisfy the constraints declared in the interface; (2) the method implementations must satisfy the specifications declared in the interface w.r.t. the symbol definitions in this class.

– **Precise specification and verification of client codes under the open-world assumption.** Under the open-world assumption, the verification of client code using interfaces can not assume what classes implement the interfaces. The advantage is that programmers can add new implementing classes without re-verifying the client code. Based on the constraints declared in the interface, client code using interfaces can be verified without knowledge about the implementing classes. Furthermore, when more information about the runtime classes is given, the specifications of the client code can be specialized to more precise ones without re-verification.

The rest part of this paper is organized as follows. Section 2 describes the syntax of the small language used in this paper. The semantics of this language is given in Section 3. The proof rules and the soundness of these rules are also given in this section. The approach to verify client code using interfaces is presented in Section 4. Section 5 concludes this paper.

2 The Syntax of the Small Language Used in This Paper

A program of the small language consists of a set of interface declarations and class definitions. An interface can be implemented by one or more classes, while a class can implement zero or more interfaces.

An interface declares a set of polymorphic function (predicate) symbols together with a set of constraints about these function (predicate) symbols. A set of methods to be invoked by its client code are also declared. For each method, the precondition and postcondition are given using the function symbols declared in this interface.

A class definition C defines a set of methods and function symbols. For each interface I implemented by the class C, all the methods and function symbols declared in I should be defined in C. The definitions of these function symbols should satisfy the constraints declared in I. For a method m declared in I, the specification of m in C is just the corresponding specification declared in the interface, w.r.t. the function symbol definitions given in C.

2.1 Types, expressions and statements

In this subsection, we describe the types, expressions, and statements associated with interfaces and classes.

2.1.1 Types

Besides the types used in Ref. [2], an interface I or a class C can also be used as a type. A value of type I is a reference to an object of some class implements I. A value of type C is a reference to an object of C. C is a subtype of I if C implementing I.

The memory layout for objects are same as the memory layout for record types in Ref. [2]: a variable with interface or class type is treated as a reference to a record type, and the member variables are treated as fields of the record type. The axioms for memory layout of record types in Ref. [2] can be adopted.

2.1.2 Expressions associated with interfaces and classes

There are several kinds of expressions associated with interfaces and classes in programs. The memory scope rules[2] for these expressions are given in Table 1.

- **this** is a keyword referring to the current object being manipulated.
- **theClass** is a keyword referring to the runtime class of **this** object.
- **classOf**(e) is the runtime class of the expression e.
- **Member variables.** For simplicity, member variables are always private. A member variables v of a class C can only occur in the method bodies and function symbol definitions in C, in the form **this**$\to v$ (or abbreviated as v).
- **ret.** It represents the return value in the post-condition of methods.
- **Applications of function symbol.** Interfaces and classes can declare (define) two kinds of function symbols: object function symbols and class function symbols (details will be given in Subsection 2.2 and 2.3).
- Let f be an object function symbol declared as $T\ f(\overline{x})$, the expression $e{\to}f(\overline{y})$ applies the definition of f in the class **classOf**(e) to the real parameters $\overline{y}$. We usually use $f(\overline{y})$ as an abbreviation for **this**$\to f(\overline{y})$.
- Let f be a class function symbol declared as $T\ f(\overline{x})$, $cexp{::}f(\overline{y})$ applies the definition of f in the class specified by $cexp$ to the real parameters $\overline{y}$. Here $cexp$ is either a class name, or the keyword **theClass**, or **classOf**(e) for some expression e. We usually use $f(\overline{y})$ as an abbreviation for **theClass**$\to f(\overline{y})$.

The function symbols and the keyword **theClass**, **ret** are used only in the specification part of programs.

Table 1 The memory scopes of expressions associated with interfaces and classes

Expression	Memory scope	Expression	Memory scope
a class name	$\emptyset$	**this**$\to v$	&**this**$\to v$
theClass, **this**	$\emptyset$	$e{\to}f(\overline{y})$	$\mathfrak{M}(e) \cup \mathfrak{M}(\overline{y}) \cup e{\to}\mathfrak{M}(f)(\overline{y})$
classOf(e)	$\mathfrak{M}(e)$	$cexp{::}f(\overline{y})$	$\mathfrak{M}(cexp) \cup \mathfrak{M}(\overline{y}) \cup cexp{::}\mathfrak{M}(f)(\overline{y})$

2.1.3 Statements associated with interfaces and classes

The following are statements associated with methods, interfaces and classes.

- **The block statement.** A block statement $\{vars, stat\}$ first allocates memory units for the variables in $vars$, then executes the statement $stat$, and finally de-allocates the memory units for $vars$. The body of each method definition must be a block statement.

- **The return statements.** The **return** statement can only appear at the end of a method body. The statement '**return** exp' first evaluates the value of exp, and then returns this value.
- **Object creation statements.** An object creation statement $v := \textbf{new}\ C(\overline{y})$ creates a new object of the class C using the real parameter $\overline{y}$, and then assigns the object reference to the variable v.
- **Method invocation statements.**

  A method invocation statement $e{\rightarrow}m(\overline{y})$ invokes the method m defined in the runtime class of e with the real-parameters $\overline{y}$. A method invocation statement $v := e{\rightarrow}m(\overline{y})$ invokes the method m and stores the return value into v.

2.2 Interface declarations

An interface declaration declares a list of function/predicate symbol, a set of constraints about the function symbols, and a set of methods.

2.2.1 Function symbols

For each function symbol, the result type, arity, and parameter types are declared. These function symbols are polymorphic and will be defined differently in the classes implementing this interface.

For each symbol f declared, the memory scope function symbol, i.e. denoted as $\mathfrak{M}(f)$, is also implicitly declared. $\mathfrak{M}(f)$ specifies the set of memory units accessed during the evaluation of f (See Ref. [2] for details).

There are two kinds of function symbols: *class symbols* (declared with the keyword **static**) and *object symbols.* Object function symbols describe properties about individual objects, while class symbols describe properties about the class.

Besides the explicitly declared function symbols, each interface has three special object function symbols: **SetOf**(**Ptr**) `BLOCK`(), **SetOf**(**Ptr**) `pmem`(), and **bool** `INV`(). Intuitively speaking, `BLOCK` yields the memory units assigned to the member variables of the object, `pmem`() yields the private memory owned by the object, and `INV`() is the invariant of the object. It is required that $o{\rightarrow}\texttt{INV}()$ holds before/after each method invocation to o.

A special kind of object symbols are called *attribute symbols* (declared with the keyword **attrib**). An attribute symbol f has no formal parameter and satisfies the following constraint.

$$\forall o : \textbf{theClass}.(o \neq \textbf{nil} \Rightarrow (o{\rightarrow}\texttt{INV}() \Rightarrow o{\rightarrow}\mathfrak{M}(f)() \subseteq \texttt{pmem}()))$$

Intuitively speaking, attribute symbols access only the private memory of the object. The function symbols `BLOCK`(), `pmem`() and `INV`() are attribute symbols.

2.2.2 Constraints about function symbols

The constraints declared in an interface are a set of formulas about the function symbols. The function symbol definitions in implementing classes must satisfy these constraints. These constraints make it possible to reason about assertions using these symbols without referring to the definitions in implementing classes.

2.2.3 Methods and their specifications

For each method declared, the method name, formal parameters, return type, preconditon, and postconditoin are given. The function symbols declared in the interface can be used in preconditions and postconditions. We use $I::m(\overline{x}) : \{P\}\_\{Q\}$ to describe that m is a method declared in an intereface I, the formal parameters, precondition, and post-condition of m are respectively $\overline{x}$, P, and Q. In P and Q, assertion variables $\rho, \rho_1, ...$ can be used to represent any assertions without the occurrence of **ret**.

Example 1. An interface `Comparable` is declared in Fig. 1. It declares two function symbols: an attribute (object) function symbol `V` and a class function symbol `LE`. The memory scope function symbols $\mathfrak{M}(\texttt{V})$ and $\mathfrak{M}(\texttt{LE})$ are also implicitly declared in `Comparable`.

Three constraints are explicitly given in this interface. These constraints specify that `LE` is a total order over integers. Together with the attribute function symbol `V`, it indirectly induces a total order over the objects. Because `V` is an attribute symbol, we have the following implicit constraint.

$$\forall o : \textbf{theClass}.(o \neq \textbf{nil} \Rightarrow (o{\rightarrow}\texttt{INV}() \Rightarrow o{\rightarrow}\mathfrak{M}(\texttt{V})() \subseteq \texttt{pmem}()))$$

There are also such implicit constraints about other attribute function symbols.

One method `compareTo` is declared. The pre-/post-conditions of `compareTo` are given using `V` and `LE`. Intuitively speaking, `compareTo` returns negative, zero, or positive integers respectively when **this** is less than, equal to, or greater than the parameter o, according to the total order induced by `V` and `LE`. In the specification, ρ is an assertion variable. It can be substituted with any assertion. Intuitively speaking, ρ in the specification means that if any assertion holds when `compareTo` is invoked, it still holds after the execution of `compareTo`, i.e. `compareTo` is a pure method. □

```
interface Comparable{
 funcs:  attrib int V();
         static bool LE(int v1, int v2);
 cons:   $\forall v : \textbf{int}.\textbf{theClass} :: \texttt{LE}(v, v)$;
         $\forall v_1, v_2 : \textbf{int}.(\textbf{theClass} :: \texttt{LE}(v_1, v_2) \vee \textbf{theClass} :: \texttt{LE}(v_2, v_1))$;
         $\forall v_1, v_2, v_3 : \textbf{int}.(\textbf{theClass} :: \texttt{LE}(v_1, v_2) \wedge \textbf{theClass} :: \texttt{LE}(v_2, v_3) \Rightarrow \textbf{theClass} :: \texttt{LE}(v_1, v_3))$;
 methods:
   int compareTo(Comparable o);
      pre  $\rho \wedge o \neq \textbf{nil} \wedge \textbf{classOf}(o) = \textbf{theClass}$
      post $\rho \wedge (\textbf{theClass} :: \texttt{LE}(\texttt{V}(), o{\rightarrow}\texttt{V}()) \Leftrightarrow \textbf{ret} \leq 0) \wedge (\textbf{theClass} :: \texttt{LE}(o{\rightarrow}\texttt{V}(), \texttt{V}()) \Leftrightarrow \textbf{ret} \geq 0)$
}
```

Figure 1. The interface `Comparable`.

2.3 Class definitions

A class definition is composed of a list of interface names implemented by this class, a list of member variables, a list of function symbol definitions, and a list of method definitions. A class can implement zero or many interfaces.

2.3.1 Member variable declarations

For each member variable declared in the class, the type and variable name are given. The member variables are private. They can only be accessed in function symbol definitions and method definitions of this class. In these definitions, a member variable v can be accessed as **this**$\rightarrow v$, or just v for abbreviation.

2.3.2 Function symbol definitions

A function symbol definition is composed of the result type T, formal parameters $\overline{x}$ and the expression e. A function definition can be written as $T\ f(\overline{x}) \triangleq e$. The definition of $\mathfrak{M}(f)$ is **Setof**(**Ptr**) $\mathfrak{M}(f)(\overline{x}) \triangleq \mathfrak{M}(e)$. Please refer to Ref. [2] for the details of memory scope expressions.

The keyword **this** and member variables are forbidden in the definitions of class function symbols, because class function symbols are not about individual objects.

The definition of `BLOCK` is derived directly from the member variable list.

$$\textbf{attrib SetOf(ptr)}\ \texttt{BLOCK()} \triangleq \{\&\textbf{this}\rightarrow v | v \text{ is a member variable}\}$$

People can also give their own definitions to `pmem`() and `INV`(), or just use the following default definitions: **attrib SetOf**(**ptr**) `pmem`() $\triangleq$ **this**$\rightarrow$`BLOCK`() and **attrib bool** `INV`() $\triangleq$ **true**.

2.2.3 Method definitions and specifications

Each method definition consists of the signature, method body, and precondition/ postcondition of this method.

The body of a method is a block statement. It is required that the method body can not assign new values to the formal parameters. For a method with a return type other than **void**, the last statement in the method body must be a **return** statement. The keyword **ret** is used in the post-condition to represent the return value.

A class must define one and only one constructor used to create new objects of this class. The constructor shares the same name with the class. The object has not been created yet when a constructor is invoked, so the keyword **this** can not occur in the precondition of a constructor. In the post-condition, the keyword **this** refers to the object created by the constructor.

For each interface I implemented by a class C, all the methods declared in I should be implemented in C. The pre-/post-condition of such a method are derived by substituting **theClass** with C in the corresponding pre-/post-conditions declared in the interface.

Example 2. The class `Point` given in Fig. 2 implements the interface `Comparable`.

Two member variables `x` and `y` are declared in `Point`.

Both the function symbols `V` and `LE` declared in `Comparable` are defined in `Point`. This class also defines two attribute functions `FldX`() and `FldY`(), which yields the value of the member variables `x` and `y`. The function symbols `pmem` and `INV` are not explicitly defined in `Point`, so the default definitions are used.

The constructor of `Point` creates a new object with `x` and `y` set to 0.

The method `compareTo` declared in `Comparable` is defined in `Point`. The precondition and postcondition of `compareTo` are derived from the corresponding specification in `Comparable` by substituting **theClass** with `Point`.

Three other methods (`Set`, `getX` and `getY`) are defined in this class. The assertion variable ρ in the specification of `Set` means that if an assertion holds when `Set` is invoked, and its memory scope is disjoint with the private memory of the object, this assertion still holds after the invocation.

Substituting **theClass** with `Point` in the constraints explicitly declared in `Comparable`, we have the following three constraints.
$\forall v : \mathbf{int}.\texttt{Point::LE}(v, v)$, $\forall v_1, v_2 : \mathbf{int}.(\texttt{Point::LE}(v_1, v_2) \vee \texttt{Point::LE}(v_2, v_1))$, and $\forall v_1, v_2, v_3 : \mathbf{int}.(\texttt{Point::LE}(v_1, v_2) \wedge \texttt{Point::LE}(v_2, v_3) \Rightarrow \texttt{Point::LE}(v_1, v_3))$. It can be checked that the definition of `Point::LE` satisfies all these constraints.

Because `V` is an attribute symbol, the constraint $\forall o : \texttt{Point}.(o \neq \mathbf{nil} \Rightarrow (o{\rightarrow}\texttt{INV}() \Rightarrow (o{\rightarrow}\mathfrak{M}(\texttt{V})() \subseteq o{\rightarrow}\texttt{pmem}())))$ must be satisfied. According to the rules given in Ref. [2], $\mathfrak{M}(\texttt{V})$ is defined as $\{\&\mathbf{this}{\rightarrow}x, \&\mathbf{this}{\rightarrow}y\}$. It can be checked that the above constraint is satisfied. All the constraints about other attribute function symbols are also satisfied. $\square$

```
class Point impl Comparable {
var:    int x, y;
funcs: attrib int V() $\triangleq x*x+y*y$;        static bool LE($v_1, v_2$) $\triangleq v_1 \leq v_2$;
       attrib int FldX() $\triangleq$ x;               attrib int FldY() $\triangleq$ y;
method:
   Point() pre $\rho$ post $\rho \wedge (\mathfrak{M}(\rho) \cap \texttt{BLOCK}() = \emptyset) \wedge$ FldX()=0 $\wedge$ FldY()=0 {x = 0; y = 0;};
   void Set(int x1, int y1) pre $\rho \wedge (\mathfrak{M}(\rho) \cap \text{pmem}() = \emptyset)$ post $\rho \wedge$ FldX()=x1 $\wedge$ FldY()=y1
       {x:=x1; y:=y1;};
   int getX() pre $\rho$ post $\rho \wedge$ (ret = FldX()) {return x;};
   int getY() pre $\rho$ post $\rho \wedge$ (ret = FldY()) {return y;};
   int compareTo(Comparable* o)
       pre   {$\rho \wedge o \neq \mathbf{nil} \wedge \mathbf{classOf}(o) = \texttt{Point}$}
       post {$\rho \wedge (\texttt{Point::LE}(\texttt{V}(), o{\rightarrow}\texttt{V}()) \Leftrightarrow \mathbf{ret} <= 0) \wedge (\texttt{Point::LE}(o{\rightarrow}\texttt{V}(), \texttt{V}()) \Leftrightarrow \mathbf{ret} \geq 0)$}
       {      int tmp1, tmp2;
              tmp1 = o→getX();          tmp2 = o→getY();
              return x*x + y*y - tmp1*tmp1 - tmp2*tmp2;
       }
}
```

Figure 2. The class `Point` implementing the interface Comparable.

2.4 Proof obligations of programs

There are two kinds of proof obligations of programs.

– The function symbol definitions in a class should satisfying the constraints declared in the interfaces implemented by this class.

– The method definitions in a class C should satisfy their specifications. Specifically, if a method m is declared in an interface I implemented by C, the specification of m in C is derived by substituting **theClass** with C.

The proof rules used to verify method definitions w.r.t. their specifications are given in the next section.

3 The Semantics and the Proof Rules for Statements

3.1 The program states

We first define some sets used to model program states. `Classes` is the set of classes defined in a program. We use $\texttt{Flds}(C)$ to denote the set of member variable names of the class C. `ObjRefs` is the unbounded set of object references of some classes in `Classes`. `Addresses` is the set of addresses of the memory units. Each memory unit can store an integer, boolean, or object reference in `ObjRefs`. The following two maps relating the elements in the above three sets.

1. $\texttt{ClsOf} : \texttt{ObjRefs} \rightarrow \texttt{Classes}$. For each object reference r in `ObjRefs`, $\texttt{ClsOf}(r)$ is the class of the object referred by r.

2. $\texttt{FldAddr} : \texttt{ObjRefs} \times \texttt{Name} \rightarrow \texttt{Addresses}$. For an object reference r in `ObjRefs` and a member variable name fn in $\texttt{Flds}(\texttt{ClsOf}(r))$, $\texttt{FldAddr}(r, fn)$ is the address of the member variable fn of the object referred by r. It is required that $\texttt{FldAddr}(r_1, fn_1) = \texttt{FldAddr}(r_2, fn_2) \Rightarrow (r_1 = r_2) \wedge (fn_1 = fn_2)$ and $r \neq nil \wedge fn \in \texttt{Flds}(\texttt{ClassOf}(r)) \Rightarrow \texttt{FldAddr}(r, fn) \neq nil$.

Definition 1. A program state s is a tuple (en, st), where

– en is called the environment, which is a partial map from variable names to addresses. Given a variable name v, $en(v)$ is the address for v. It is required that **this** and **ret** are always in the domain of en,

– st is called the store, which is a partial map from ***Addresses*** to ***boolean*** $\cup$ ***integers*** $\cup$ *ObjRefs*.

Given a program state (en, st), we use $\texttt{ref}_{st}$ to denotes all the object references in the co-domain of st. For each object reference r in $\texttt{ref}_{st}$ and a member variable fn of $\texttt{ClsOf}(r)$, it is required that $\texttt{FldAddr}(r, fn)$ is in $\texttt{Dom}(st)$.

Given two maps m_1 and m_2, $m_1 \dagger m_2$ is the map satisfying $(m_1 \dagger m_2)(x) = m_2(x)$ if $x \in \texttt{Dom}(m_2)$, $(m_1 \dagger m_2)(x) = m_1(x)$ otherwise. Given a map m and a set s, $s \lhd m$ denotes the map $\{d \mapsto m(d) | d \in (\texttt{Dom}(m) \cap s\}$. Given two vectors $\overline{v}_1, \overline{v}_2$ of same length n, $\{\overline{v}_1 \mapsto \overline{v}_2\}$ denotes the map $\{\overline{v}_1[i] \mapsto \overline{v}_2[i] | i = 1, \ldots, n\}$.

Given a program state (en, st) and a vector $\overline{names}$ of names, and a vector $\overline{values}$ of values corresponding to $\overline{names}$, $(en, st) \uplus (\overline{v}, \overline{values})$ denotes the program state $(en \dagger \{\overline{names} \mapsto \overline{addr}\}, st \dagger \{\overline{addr} \mapsto \overline{values}\})$, where $\overline{addr}$ is a vector of addresses satisfying $\overline{addr} \cap \texttt{Dom}(st) = \emptyset$. Intuitively speaking, $(en, st) \uplus (\overline{names}, \overline{values})$ is the program state after a set of memory units are allocated for the variables in $\overline{names}$, and assigned the values in $\overline{values}$.

3.2 Semantics of expressions

The semantic of expressions are maps from program states to values. Given a program state (en, st), we write $[e]^{st}_{en}$ to denote the value of an expression e at (en, st). Because of space limitation, we just give the semantics of expressions associated with interfaces and classes in Table 2.

The semantics of $e{\to}f(\overline{y})$ and $cexp{::}f(\overline{y})$ shows that function symbols are polymorphic, i.e. different definitions of f are referred according to the runt-time class of e and $cexp$.

The semantics of the special operator & is also given here. It yields the left value (i.e. address) of its operand. This operator is only used in specifications.

Table 2 The semantics of some expressions

e	$[e]^{st}_{en}$	e	$[e]^{st}_{en}$
ret	$st(en(\mathbf{ret}))$	$e{\to}f(\overline{y})$	$[e_{def}]^{st'}_{en'}$, where e_{def} is the definition of f in $\mathtt{ClsOf}([e]^{st}_{en})$ $(en', st') = (en, st) \uplus (\mathbf{this}, [e]^{st}_{en}) \uplus (\overline{x}, \overline{y})$ $\overline{x}$ is the formal parameters of f
this	$st(en(\mathbf{this}))$	$cexp{::}f(\overline{y})$	$[e_{def}]^{st'}_{en'}$ where e_{def} is the definition of f in $[cexp]^{st}_{en}$ $(en', st') = (en, st) \uplus (\overline{x}, \overline{y})$ $\overline{x}$ is the formal parameters of f
theClass	$\mathtt{ClsOf}([\mathbf{this}]^{st}_{en})$	$\mathbf{this} \to v$	$st(\mathtt{FldAddr}([\mathbf{this}]^{st}_{en}, v))$
ClassOf(e)	$\mathtt{ClsOf}([e]^{st}_{en})$		
$\&v$	$en(v)$	$\&(e{\to}n)$	$\mathtt{FldAddr}([e]^{st}_{en}, n)$

3.3 Semantics and proof rules of program statements

The semantics of of a statement $stat$, denoted as $[stat]$, is a partial map from program states to program states.

Statements are specified using Hoare's Triples of the form $\{P\}\ stat\ \{Q\}$, which means that if P holds at a program state s, $s' = [stat](s)$, then Q holds at s'. In the post-condition Q, $\overleftarrow{e}$ is used to denote the value of e evaluated at the pre-state s. So e in P and $\overleftarrow{e}$ in Q denote the same value. Because of space limitation, we only give the semantics and proof rules (depicted in Figure 3) of statements associated with interfaces or classes.

3.3.1 The return statements

The statement '**return** e' evaluates the value of exp and then returns this value. The semantic of '**return** e' is formally defined as

$$[\mathbf{return}\ e](en, st) \triangleq (en, st \dagger \{en(\mathbf{ret}) \mapsto [e]^{st}_{en}\})$$

It can be checked that the axiom RETURN-ST in Fig 3 is sound.

3.3.2 Block statement

A block statement is of the form $\{vars, stat\}$, where $vars$ is a list of variables, and $stat$ is a statement. When such a block is executed, the memory units for variables in $vars$ are firstly allocated, then $stat$ is executed, finally the memory units for $vars$ are de-allocated. The semantic of this block is formally defined as $[\{vars, stat\}](en, st) \triangleq (en, st'')$ where $(en', st') = [stat]((en, st) \uplus (vars, -))$ and $st'' = (\mathtt{Dom}(st'') - en'[vars]) \lhd st'$.

The soundness of the rule BLOCK-ST. Let `LocMem` be the expression $\{\&v | v \text{ is a variable in } vars\}$. Suppose that an assertion P holds at (en, st), $P \wedge (\mathfrak{M}(P) \cap \texttt{LocMem} = \emptyset)$ holds at the state $(en, st) \uplus (vars, -)$ because `LocMem` are not in $\texttt{Dom}(st)$ and no variable in $vars$ occurs in P. If the premises of BLOCK-ST holds, $Q \wedge (\mathfrak{M}(Q) \cap \texttt{LocMem} = \emptyset)$ holds at (en', st'). So Q holds at (en, st'') because no variable in $vars$ occurs in Q and $\mathfrak{M}(Q) \cap \texttt{LocMem} = \emptyset$.

RETURN-ST:$\{Q[e/\mathbf{ret}]\}$ **return** e $\{Q\}$

$$\boxed{\text{BLOCK-ST}^1}\ \frac{\{P \wedge (\mathfrak{M}(P) \cap \texttt{LocMem} = \emptyset)\}\ stat\ \{Q \wedge (\mathfrak{M}(Q) \cap \texttt{LocMem} = \emptyset)\}}{\{P\}\ \{vars; stat\}\ \{Q\}}$$

$$\boxed{\text{CSTOR-SPEC}^2}\ \frac{\begin{array}{l}\{P \wedge \mathbf{this} \neq \mathbf{nil} \wedge \mathbf{classOf}(\mathbf{this}) = C \wedge (\mathbf{this}{\rightarrow}\texttt{BLOCK}() \cap \mathfrak{M}(P) = \emptyset)\} \\ \texttt{Block}\ \ \{Q\}\end{array}}{C{::}C(\overline{x}) : \{P\}\_\{Q\}}$$

$$\boxed{\text{OBJ-CREATION}}\ \frac{\mathrm{C}{::}\mathrm{C}(\overline{x}) : \{P\}\_\{Q\}}{\{P[\overline{y}/\overline{x}]\}\ \ v = \mathbf{new}\ \mathrm{C}(\overline{y})\ \ \{(\mathfrak{M}(Q[\overleftarrow{\overline{y}}/\overline{x}]) \cap \{\&v\} = \emptyset) \Rightarrow Q[\overleftarrow{\overline{y}}/\overline{x}][v/\mathbf{this}]\}}$$

$$\boxed{\text{METHOD-SPEC}^2}\ \frac{\{P \wedge \mathbf{this} \neq \mathbf{nil} \wedge \mathbf{classOf}(\mathbf{this}) = C\}\ \ \texttt{Block}\ \ \{Q\}}{C{::}m(\overline{x}) : \{P\}\_\{Q\}}$$

$$\boxed{\text{INVOC-1}^3}\ \frac{I{::}m(\overline{x}) : \{P\}\_\{Q\}}{\{(e \neq \mathbf{nil}) \wedge P'\}\ v = e{\rightarrow}m(\overline{y})\ \{(\mathfrak{M}(Q') \cap \{\&v\} = \emptyset) \Rightarrow Q'[v/\mathbf{ret}]\}}$$

$$\boxed{\text{INVOC-2}^3}\ \frac{I{::}m(\overline{x}) : \{P\}\_\{Q\}}{\{(e_1 \neq \mathbf{nil}) \wedge P'\}\ e{\rightarrow}m(\overline{y})\ \{Q'\}}$$

Notes:
1. No variable in $vars$ occurs in P, Q. `LocMem` is the abbreviation for $\{\&v|v \text{ is a variable in } vars\}$.
2. `Block` is the body of the constructor of C.
3. I is the static type of e, P' is an abbreviation for $P[\mathbf{classOf}(e)/\mathbf{theClass}][\overline{y}/\overline{x}][e/\mathbf{this}]$, Q' is an abbreviation for $Q[\mathbf{classOf}(\overleftarrow{e})/\mathbf{theClass}][\overleftarrow{\overline{y}}/\overline{x}][\overleftarrow{e}/\mathbf{this}]$

Figure 3. The axioms and proof rules for some program statements.

3.3.3 Rules for specification and invocation of object constructors

A constructor specification $C{::}C(\overline{x}) : \{P\}\_\{Q\}$ means that if this constructor is invoked with real parameters $\overline{x_0}$ at a state satisfying $P[\overline{x_0}/\overline{x}]$, a new C object is created, and the state after the invocation satisfying $Q[\overline{x_0}/\overline{x}]$.

The soundness of the rule CSTOR-SPEC. Suppose that the constructor $C{::}C$ is invoked at a program state (en, st). Firstly, the memory units for **this** and the formal parameters are allocated, a new object r ($\mathbf{clsOf}(r) = C$ and $r \notin \texttt{ref}_{st}$) and the corresponding real parameters $\overline{x_0}$ are assigned to these memory units. Now the program state is $(en', st') = (en, st) \uplus (\overline{x}, \overline{x_0}) \uplus (\mathbf{this}, r)$, where $r \notin \texttt{ref}_{st} \wedge (r{\rightarrow}\texttt{BLOCK}() \cap \texttt{Dom}(st) = \emptyset)$. The execution of the constructor body `Block` changes the program state to $[\texttt{Block}](en', st')$. The memory units for formal parameters is finally de-allocated.

If $P[\overline{x_0}/\overline{x}]$ holds at (en, st), it still holds at (en', st') because $\overline{x}$ and **this** do not occur in $P[\overline{x_0}/\overline{x}]$. At the state (en', st'), the values of $\overline{x}$ are just $\overline{x_0}$, so P holds at

(en', st'). Further more, $\mathbf{this} \neq \mathbf{nil} \wedge \mathbf{classOf}(\mathbf{this}) = C \wedge (\mathbf{this} \rightarrow \texttt{BLOCK}() \cap \mathfrak{M}(P) = \emptyset)$ also holds at this state. If the premise of CSTOR-SPEC holds, the assertion Q holds at $[\texttt{Block}](en', st')$. Because the values of formal parameters $\overline{x}$ are not modified by `Block`, $Q[\overline{x_0}/\overline{x}]$ also holds at $[\texttt{Block}](en', st')$. $Q[\overline{x_0}/\overline{x}]$ still holds after the memory units for formal parameters are de-allocated because no formal parameter in $\overline{x}$ occurs in $Q[\overline{x_0}/\overline{x}]$.

The soundness of the rule OBJ-CREATION. The execution of an object creation statement $v := \mathbf{new}\ C(\overline{y})$ is as follows. It first invokes the constructor with the value of $\overline{y}$ evaluated at a state (en, st) as real parameters, resulting in a program state (en', st'), then assigns the new object reference (`this`) to v, and de-allocates the memory unit for **this**, resulting in a state (en, st''). Suppose that $C{::}C(\overline{x}) : \{P\}\_\{Q\}$ and $P[\overline{y}/\overline{x}]$ holds at (en, st), $Q[\overleftarrow{\overline{y}}/\overline{x}]$ holds at (en', st'). Because v does not occur in $Q[\overleftarrow{\overline{y}}/\overline{x}]$ and **this** does not occur in $Q[\overleftarrow{\overline{y}}/\overline{x}][v/\mathbf{this}]$, $Q[\overleftarrow{\overline{y}}/\overline{x}][v/\mathbf{this}]$ holds at (en, st'') if $(\mathfrak{M}(Q[\overleftarrow{\overline{y}}/\overline{x}]) \cap \{\&v\} = \emptyset)$ holds at (en', st').

3.3.4 Rules for specifications and invocations of methods

A method specification $C{::}m(\overline{x}) : \{P\}\_\{Q\}$ means that if $C{::}m$ is invoked at a program state satisfying $P[\mathbf{this}_0/\mathbf{this}][\overline{x_0}/\overline{x}]$, it results in a program state $Q[\mathbf{this}_0/\mathbf{this}][\overline{x_0}/\overline{x}]$.

The soundness of the rule METHOD-SPEC. Suppose that the method $C{::}m$ is invoked with real parameters $\overline{x_0}$ to an object $\mathbf{this}_0$ ($\mathbf{this}_0 \neq \mathbf{nil} \wedge \mathbf{classOf}(\mathbf{this}_0) = C$) at a program state (en, st) satisfying $P[\mathbf{this}_0/\mathbf{this}][\overline{x_0}/\overline{x}]$. First, the memory units for **this** and the formal parameters $\overline{x}$ is allocated and assigned corresponding values, resulting in a program state $(en', st') = (en, st) \uplus (\overline{x}, \overline{x_0}) \uplus (\mathbf{this}, \mathbf{this}_0)$. Because the values of $\overline{x}$ and **this** are respectively $\overline{x_0}$ and $\mathbf{this}_0$ at (en', st'), $P \wedge \mathbf{this} \neq \mathbf{nil} \wedge \mathbf{classOf}(\mathbf{this}) = C$ holds at (en', st'). Then the body `Block` of $C{::}m$ is executed, resulting in $(en'', st'') = [\texttt{Block}](en', st')$. Q holds at (en'', st'') if the premise of METHOD-SPEC holds. Because the values of $\overline{x}$ and **this** are still $\overline{x_0}$ and $\mathbf{this}_0$ at (en'', st''), $Q[\mathbf{this}_0/\mathbf{this}][\overline{x_0}/\overline{x}]$ holds at (en'', st''). The memory units for **this** and formal parameters are finally de-allocated. $Q[\mathbf{this}_0/\mathbf{this}][\overline{x_0}/\overline{x}]$ still holds at the final state.

The soundness of INVOC-1 and INVOC-2. Let e be an expression with static type I, $v = e \rightarrow m(\overline{y})$ invokes the method m defined in the run-time class of e with the real parameters $\overline{y}$, and assign the return value to the variables v.

Suppose that the specification of $I{::}m$ is $I{::}m(\overline{x}) : \{P\}\_\{Q\}$. The pre-/post-condition of m defined in $\mathbf{classOf}(e)$ are respectively $P[\mathbf{classOf}(e)/\mathbf{theClass}]$ and $Q[\mathbf{classOf}(\overleftarrow{e})/\mathbf{theClass}]$. If $v = e \rightarrow m(\overline{y})$ executes at a program state (en, st) satisfying $\{(e \neq \mathbf{nil}) \wedge P[\mathbf{classOf}(e)/\mathbf{theClass}][e/\mathbf{this}][\overline{y}/\overline{x}]\}$, the state (en', st') after the invocation of the method m in $\mathbf{classOf}(e)$ should satisfy $Q[\mathbf{classOf}(\overleftarrow{e})/\mathbf{theClass}][\overleftarrow{e}/\mathbf{this}][\overleftarrow{\overline{y}}/\overline{x}]$. After the value of **ret** is assigned to v, we have $Q[\mathbf{classOf}(\overleftarrow{e})/\mathbf{theClass}][\overleftarrow{e}/\mathbf{this}][\overleftarrow{\overline{y}}/\overline{x}][v/\mathbf{ret}]$ holds at at (en', st') if $\mathfrak{M}(Q[\mathbf{classOf}(\overleftarrow{e})/\mathbf{theClass}][\overleftarrow{e}/\mathbf{this}][\overleftarrow{\overline{y}}/\overline{x}]) \cap \{\&v\} = \emptyset$. So INVOC-1 is sound.

Similarly, the proof rule INVOC-2 is sound.

These two rules can still be applied if the static type of e is a class C, because **theClass** in the specification of $C{::}m$ is just C.

4 Code Verification under the Open-world Assumption

Using the proof rules INVOC-1 and INVOC-2, the specifications of method invocation statements can be derived without knowing the exact dynamic class of the receiving objects. Though the function symbols in these specifications are polymorphic and defined differently in different implementing classes, the definitions must satisfy the constraints declared in interfaces. Suppose that the static type of e is an interface I, and *constr* is a constraint in I, we have

$$(e \neq \mathbf{nil}) \Rightarrow constr[\mathbf{classOf}(e)/\mathbf{theClass}]$$

Using such properties and the specifications derived using INVOC-1 and INVC-2, we can verify client code using interfaces under the open-world assumption.

Example 3. Part of the class `arrayList` is given in Fig. 4. An `arrayList` object stores some `Comparable` objects in the array-typed member variable `a`. The specification of the method `sort` says that if all the elements of `a` are not **nil** and refer to objects of the same class, `sort` can sort these objects w.r.t. the order induced by `LE` and `V` defined in the class **classOf**(`a`[0]).

The predicate `MemLayout` specify that the private memory of each `Comparable` object is disjoint with the array `a`. So assignments to `a` do not modified the attribute `V`() of objects because `V` is an attribute symbol of `Comparable`.

Because of the space limitation, we just briefly show how to prove that the following formula is an invariant of the inner while-statement.

$$\begin{array}{l}\forall x \in (0..j).(\mathbf{CLS}::\mathtt{LE}(\mathtt{a}[x]\to\mathtt{V}(), \mathtt{a}[j]\to\mathtt{V}())) \wedge \mathtt{MemLayout}()\wedge \\ \forall x \in (0..9).(\mathtt{a}[x] \neq \mathbf{nil}) \wedge \forall x \in (0..9).(\mathbf{classOf}(\mathtt{a}[x]) = \mathbf{CLS})\end{array} \tag{1}$$

where **CLS** is the abbreviation for **classOf**(`a`[0]@1), and `a`[0]@1 is the value of `a`[0] evaluated when the program begins.

Formula 1 still holds after the assignment to `cR` (line 18). From Formula 1, **classOf**(`a`[j]) and **classOf**(`a`[j + 1]) are both **CLS**. Based on the specification of `compareTo` and the rule INVOC-1, the following formula also holds after line 18.

$$\begin{array}{l}(\mathbf{CLS}::\mathtt{LE}(\mathtt{a}[\mathrm{j}]\to\mathtt{V}(), \mathtt{a}[\mathrm{j}{+}1]\to\mathtt{V}()) \Leftrightarrow \mathrm{cR} \leq 0)\wedge \\ (\mathbf{CLS}::\mathtt{LE}(\mathtt{a}[\mathrm{j}{+}1]\to\mathtt{V}(), \mathtt{a}[\mathrm{j}]\to\mathtt{V}()) \Leftrightarrow \mathrm{cR} \geq 0)\end{array} \tag{2}$$

Because `a`[0]@1 is not **nil**, and **CLS** is a class implementing `Comparable`, substituting **theClass** with **CLS** in the constraints in `Comparable`, we have

$$\begin{array}{l}\forall v_1, v_2, v_3 : \mathbf{int}.(\mathbf{CLS}::\mathtt{LE}(v_1, v_2) \wedge \mathbf{CLS}::\mathtt{LE}(v_2, v_3) \Rightarrow \mathbf{CLS}::\mathtt{LE}(v_1, v_3)) \\ \forall v_1, v_2 : \mathbf{int}.(\mathbf{CLS}::\mathtt{LE}(v_1, v_2) \vee \mathbf{CLS}::\mathtt{LE}(v_2, v_1)).\end{array}$$

From these constraints, Formula 1 and 2, it holds that

$$\begin{array}{l}(\mathrm{cR} \leq 0)?\ \forall x \in (0..\mathrm{j}+1)\mathbf{CLS}::\mathtt{LE}(\mathtt{a}[x]\to\mathtt{V}(), \mathtt{a}[\mathrm{j}+1]\to\mathtt{V}()) \\ \qquad : \forall x \in (0..\mathrm{j}+1)(\mathbf{CLS}::\mathtt{LE}(\mathtt{a}[x]\to\mathtt{V}(), \mathtt{a}[\mathrm{j}]\to\mathtt{V}()))\end{array}$$

After line (19) swaps $a[j]$ and $a[j+1]$ if $a[j+1]$ is 'less' than $a[j]$, all the objects from $a[0]$ to $a[j]$ is less than or equal to $a[j+1]$. So the first conjunct of Formula 1 holds after j is increased by line (20).

It can also be verified that the other conjuncts of Formula 1 hold after line (20). So Formula 1 is an invariant of the inner loop. $\square$

After a piece of client code using interfaces is verified, more precise specifications can be derived without re-verification when more knowledge about context of the client code is known. Specifically, if we know that the runtime class of an expression e is a class C, we can substitute **classOf**(e) with C in the client code specification. We also know that the function symbol f in $e{\rightarrow}f$ refers to the definition of f in C.

```
1)   class arrayList{
2)   var: Comparable a[10];
3)   funcs: SetOf(Ptr) pmem() ≜ λx.(&a[x])[0..9];
4)     Comparable get(int i) ≜ a[i];
5)     bool MemLayOut() ≜ ∀i∈(0..9)(∀j∈(0..9)(&a[j] ∉ get(i)→pmem()))
6)   method:
7)     . . .
8)     void Sort()
9)     Pre    $\rho \wedge (\mathfrak{M}(\rho) \cap \mathtt{pmem}() = \emptyset) \wedge \mathtt{MemLayout}() \wedge (\forall x \in (0..9)(\mathtt{get}(i) \neq \mathbf{nil}))$
10)           $\wedge(\forall x \in (0..9)(\mathbf{classOf}(\mathtt{get}(i)) = \mathbf{classOf}(\mathtt{get}(0))))$
11)    Post   $\rho \wedge (\mathbf{classOf}(\mathtt{get}(0)) = \mathbf{classOf}(\overleftarrow{\mathtt{get}(0)}))$
12)           $\wedge\forall i \in (0..8)(\mathbf{classOf}(\mathtt{get}(0)) :: \mathtt{LE}(\mathtt{get}(i){\rightarrow}\mathtt{V}(), \mathtt{get}(i+1){\rightarrow}\mathtt{V}()))$
13)      { int i,j,cR; Points tmp;
14)        i = 9;
15)        while(i>0){
16)          j = 0;
17)          while (j<i-1)
18)          { cR = a[j]→compareTo(a[j+1]);
19)            if(cR > 0){temp = a[j]; a[j]=a[j+1]; a[j+1]=temp;} else skip;
20)            j = j+1;
21)          }
22)          i = i-1;
23)        }
24)      }
25)  }
```

Figure 4. The sort algorithm for Comparable objects.

Example 4. Suppose that `al` is non-nil and refers to an `arrayList` object. According to the specification of `arrayList::Sort`(), the following formula holds after a statement `al`→`Sort`().

$$\forall i \in (0..8)(\mathbf{classOf}(\overleftarrow{\mathtt{al}{\rightarrow}\mathtt{get}(0)}) :: \mathtt{LE}(\mathtt{al}{\rightarrow}\mathtt{get}(i){\rightarrow}\mathtt{V}(), \mathtt{al}{\rightarrow}\mathtt{get}(i+1){\rightarrow}\mathtt{V}()))$$

If all the objects in `al` are `Point` objects, we can have the following precise post-condition without re-verifying `arrayList::Sort`.

$$\forall i \in (0..8)(\mathtt{Point} :: \mathtt{LE}(\mathtt{al}{\rightarrow}\mathtt{get}(i){\rightarrow}\mathtt{V}(), \mathtt{al}{\rightarrow}\mathtt{get}(i+1){\rightarrow}\mathtt{V}()))$$

That is, the `Point` objects in the list `al` are sorted according to their distance from the original point.

5 Related Works and Conclusions

The main challenge to specify ‘programming to interface’ code is to deal with the polymorphism caused by dynamic-binding and avoid re-verification of client code using interfaces. It is also important to make client code using interfaces fulfill different function features using different implementations of the interfaces.

Many research works[3,4,5,6] have been proposed to deal with the polymorphism caused by inheritance and method overriding. Most of the works use the LSP (Liskov Substitution Principle) subtyping rule[1] to avoid re-verification of the client code. Once a method has committed to a pre-conditoin/post-condition contract, any redefinition of this method through overriding must preserve to this commitment. As we discussed before, such approaches are not suitable for the 'programming to interfaces' paradigm.

In Refs. [7,8], a lazy form of behavioral sub-typing is presented. The behaviors of the overriding methods are only required to preserve the 'part' of the specifications that actually used to verify the client codes. This approach is not suitable for the situations where programmers make a piece of client code fulfill different functional features using different implementations of an interface.

In Ref. [9], abstract attribute symbols are also used in method specifications. Because no constraint about these attributes is given, their method has a weaker capability to verify client codes using interfaces under the open-world assumption.

In this paper, we present a flexible and precise approach to specify and verify code written in the 'programming to interfaces' paradigm. An interface can be specified by a set of abstract and polymorphic function/predicate symbols together with a set of constraints. A class implementing an interface can give its own definitions to the function/predicate symbols declared in the interface, as long as the constraints declared in the interface are satisfied. The method definitions in this class must satisfy the specification given in the interface. Based on the above class-interface-implementation relations, the client code using interfaces can be verified without knowing the implementing classes of the interfaces. Furthermore, when more information about the dynamic classes of expressions are known, the client code specifications can be specialized to more precise ones without re-verification. So the approach presented in this paper can take the advantages of the 'programming-to-interfaces' paradigm.

Though class inheritance is not directly discussed in this paper, polymorphism caused by class inheritance can also be dealt with by the approach in this paper. The super-class can be viewed as an interface together with an implementation, and the sub-class viewed as another implementation of the interface.

Conflict of interest

The authors declare that they have no conflict of interest.

References

[1] Liskov BH, Wing JM. A behavioral notion of subtyping. ACM Trans. Program. Lang. Syst., November 1994, 16(6): 1811–1841.

[2] Zhao JH, Li XD. Scope logic: An extension to hoare logic for pointers and recursive data structures. In Liu ZM, Woodcock J, Zhu HB, eds. ICTAC, volume 8049 of Lecture Notes in Computer Science. Springer, 2013. 409–426.

[3] Poetzsch-Heffter A, Müller P. A programming logic for sequential java. In Swierstra SD, ed. ESOP, volume 1576 of Lecture Notes in Computer Science. Springer, 1999. 162–176.

[4] Chin W-N, David C, Nguyen HH, Qin SH. Enhancing modular oo verification with separation

logic. In Necula GC, Wadler P, eds. POPL. ACM. 2008. 87–99.

[5] Parkinson MJ, Bierman GM. Separation logic, abstraction and inheritance. Proc. of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '08. New York, NY, USA. ACM. 2008. 75–86.

[6] Smans J, Jacobs B, Piessens F, Schulte W. Automatic verification of java programs with dynamic frames. Formal Asp. Comput., 2010, 22(3-4): 423–457.

[7] Dovland J, Johnsen EB, Owe O, Steffen M. Lazy behavioral subtyping. J. Log. Algebr. Program., 2010, 79(7): 578–607.

[8] Dovland J, Johnsen EB, Owe O, Steffen M. Incremental reasoning with lazy behavioral subtyping for multiple inheritance. Sci. Comput. Program., 2011, 76(10): 915–941.

[9] Liu YJ, Qiu ZY. A Separation Logic for OO Programs. Proc. of the 7th International Conference on Formal Aspects of Component Software. FACS10. Berlin, Heidelberg. 2012. 88–105.