A Semantics of the Separation of Interface and Implementation in C++

Eun-Sun Cho
Department of Computer Science
Seoul National University
Seoul, Korea 151-742

Sang-Yong Han
Department of Computer Science
Seoul National University
Seoul, Korea 151-742

Hyoung-Joo Kim
Department of Computer Engineering
Seoul National University
Seoul, Korea 151-742

e-mail:{eschough@candy, syhan@pandora, hjk@papaya}.snu.ac.kr

Abstract

C++ uses 'class' as the basis of 'subtype-polymorphism' and 'inheritance', but it has been pointed out that the overloading of 'class' limits the expressiveness and makes its type system inflexible. This made C++ and some other object-oriented languages to separate a class into two modules – an interface and an implementation.

But, there seems to be no leading C++ model for separating interface lattice from implementation lattice. Moreover none of proposed models describe the result of the separation in a formal way. As a result it is hard to understand what the type space would be like after the separation.

This paper[1] presents a formal model for the separation of interface and implementation in C++, which explains the properties of the resulting type space after the separation.

1 Introduction

C++ uses '`public`', '`private`' or '`protected`' to control access to members or member functions of a class. When a member or a member function of a class is specified as `public`, its name is allowed to be used by any functions. But with `private`, the name is allowed to be accessed directly only by member functions of the class in which it is being declared[2].

It, however, is natural to divide members and member functions into two groups, according to whether it has the `public` keyword or not. The declarations of members or member functions which are specified as `public` are called an '*interface*'. Note that only through the *interface* of a class, users can be informed of all about what the class provides, while the other members or member functions of a class cannot be accessed by the users of the class. Instead, they are responsible for implementing the class. So, we call all definitions of members and member functions including non-public ones '*implementation*' of the class.

The concept of *interface* and *implementation* is melted into a 'class' in C++. It is noticeable that their corresponding hierarchies yield some problems. For example, as aggregation types such as class '*Set*' and class '*Bag*' have common properties, they may have same declarations of public member functions - *insert, delete* etc. Thus, such declarations may be retrieved into an abstract class '*Collection*', and all aggregation classes may be derived from it. However, it is likely that class '*Bag*' would be derived from class '*Bptree*' or something, and class 'Set' would be derived again from the class '*Bag*'. So, the inheritance hierarchy in C++ should imply two independent hierarchies as figure 1 shows.

Moreover, class '*Set*' also can be made an abstract class, and drive class '*Bptree*', class '*Hash*', and class '*BitString*' from it, since a set can be implemented in several ways including B+tree or Hash. But such schema may increase complexity of class hierarchies or may be impossible to be described because of the complexity.

[1]This work is supported in part by KOSEF under grant KOSEF94-2180, "Research on Object-Oriented Database Programming Language Based on C++ and ODMG Standard Object Model".

[2]`friend` and `protected` in C++ are not considered in this paper.

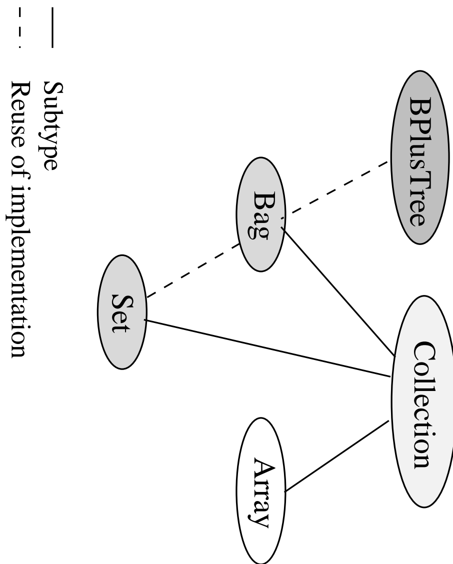


Figure 1: Two hierarchies including class *Set*

2 Related Works

Many object-oriented language designers, pointing out these problems, have concentrated on the separation of interface and implementation, and tried to provide users with the separated hierarchies - one for subtyping and the other for reuse. From now on, we'll call the *s*eparation of *i*nterface and *i*mplementation, 'SII', in abbreviation.

However each existing mechanism for SII is accompanied by its own semantics of the type system, and there are a lot of different approaches to processing SII[2, 3, 8, 10, 11, 12, 13, 14]. In some languages[2, 10, 12, 13], an interface is allowed to be bound to more than one implementation, while only one implementation is allowed in some languages. In some language definitions, there is no concept of the hierarchy of either types or implementations[13, 14].

Some languages provides users with the features for specifying the type hierarchies explicitly[9, 14, 16], while in others, the compiler finds the dependency of the conformance among the types to define a hierarchy by comparing the structures of their interfaces[13]. Like subtyping, bindings of an interface and implementations are to be done explicitly[12, 14], or implicitly[2, 10, 13].

As for C++, it already supports implementation inheritance hierarchies which are specified explicitly by user. And in most suggestions for SII in C++, one interface can have more than one implementation.

2.1 SII in C++

If the features for SII are supported in an C++ extension, users may bind an interface and implementations when declaring implementation classes and declaring and using their objects.

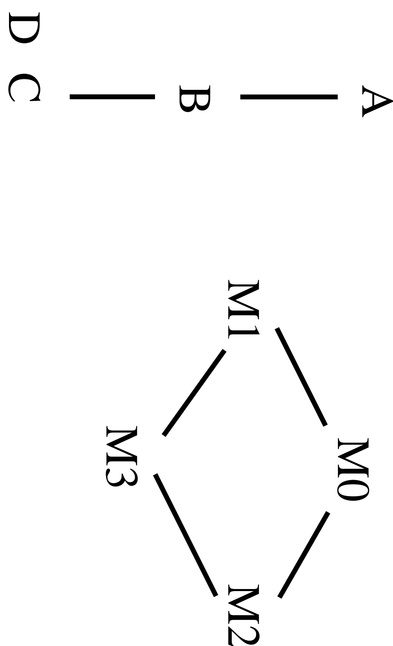
First, when an implementation class is being declared, the declaration of an implementation class can include its corresponding interface in either an explicit or implicit way. As in figure 3, binding is done explicitly with the keyword '`implements`'.

When an object is created by the operator '`new`' with the name of an implementation, it may be assigned to a pointer of an interface to which the implementation was bound in the declaration time. For example, '`Set * p = new _BptreeSet;`' generates an object whose implementation is class *_BptreeSet* and assigns it to a pointer of interface *Set*. Through the pointers of an interface, like '`p`', all members and member functions are able to be accessed without concerning about the representation and implementation of the actual object.

2.1.1 SII suggested by Martin et al.[23]

A model was presented for C++ extensions to support SII in Martin et al.[12]. Each interface and implementation is transformed into a C++ class by the preprocessor. The C++ class which represents an implementation is combined with an interface by inheriting the C++ abstract class transformed from the interface. Thus, an implementation class is transformed to be a C++ class derived from its interface and base classes at once.

Figure 2 shows an example of hierarchies. Class *A*–*B*–*C* represent a hierarchy of interfaces, and class *D* is an interface that is not in the hierarchy. Implementations *M1, M2* are derived from *M0*, and *M3* from *M1, M2*. *M1, M2* and *M3* are explicitly bound to *A, C* and *D* respectively. User-defined separated hierarchies are in (a), and the resulting class hierarchy after transformation is in (b). The solid lines in (b) are same as those in (a), while the dotted lines represent the bindings of interfaces and implementations. Though this mechanism is simple and intuitive, it has some limitations – for example, in figure 2, it is not clear if *M3* conforms to the interface *A* and *C*. This kind of binding is not specified by users, but introduced by the side-effects produced by the derivation of implementations.



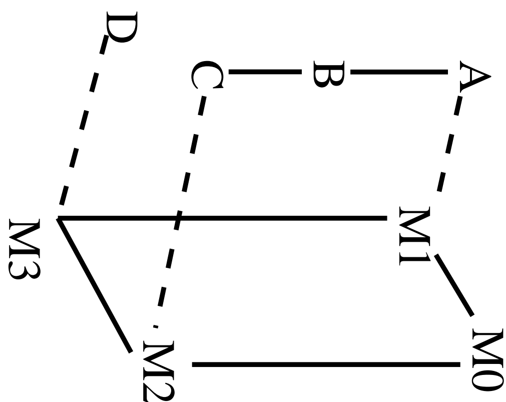


Figure 2: hierarchies of interfaces, implementations and the resulting type space

2.1.2 SII suggested by Granston et al.[17] and Baumgartner et al.[1]

At Purdue University, C++ extension for SII has been investigated from 1991[2, 10].

In [2, 10], interfaces are denoted by '`signature`'. And bindings of a signature and implementations are not specified explicitly. Instead, the complier computes their relationships by comparing the structural definitions of signatures and implementations, which is called 'structural conformance checking[6]'. Moreover, the subtype relationships for signatures are also inferred in a similar way.

However, from these properties, some problems may exist in its recursive types and covariance rules in signature inheritance which are not allowed in C++[1, 4, 7]. And unrecognized relationships among interfaces are likely to be made, since meaningful interface hierarchies by users are jumbled with other compiler-generated relationships among interfaces[15].

As described in above, mechanisms to separate and combine interfaces and implementations in C++ are based on different models. And each of them has no uniform semantics, because SII was introduced in a feature-to-feature way. In this paper, we present a new model for SII in C++ which provides a uniform semantics without any problems described in the previous sections.

3 The Proposed Model

3.1 Introduction

Like others, our model also has two axes – 'interface' and '(implementation) class'. Member declarations are allowed to be in an interface, for the users who want to access data directly. An implementation is bound explicitly to an interface, using the '`implements`' keyword which is illustrated in figure 3.

The declarations of members or member functions are omitted in the declaration of an implementation class if they are already in the declaration of its corresponding interface. In an interface definition, only interface names are allowed to be used to declare members and member functions, so the world of interfaces is *self-contained.*

A members or member functions in the declaration of an interface can be redefined with an implementation type in a declaration of corresponding implementations. For example, suppose 'A' is an interface and

```
interface Set{
    int insert(); };

class _BptreeSet{
   implements  Set;  // 'int insert()' included
   int tree_depth; };

// Use the interface and implementation
Set * a = new _BptreeSet;
a->insert(); b.insert();
```

Figure 3: An example of declarations and uses of an interface and an implementation

'*M1*' is one of its implementations. If '*A*' has a member declaration, say, '`B x;`' and '*B*' is the interface of '*M2*', then '`x`' can be redefined in '*M1*' with type '*M2*'.

A class '*C*' conforms to an interface 'S' when (1) the interface bound to '*C*' is a subtype of the interface '*S*', or (2) the smallest super class of '*C*' which is bound to an interface explicitly, conforms to the interface '*S*'.

3.2 Formal Definitions

As we described above, the semantics of SII in Martin et al.[12] can be explained only by the resulting multiple inheritance hierarchies[12]. And though the models in Gnu C++[2, 10] takes a few of the existing object-oriented semantics from various sources, the absence of uniform semantics causes conflicts as described in the previous section.

In this section, we concentrate on the formal semantics of the proposed model.

3.2.1 Algebraic definitions

Basic Sets

Both interfaces and implementations are considered as the lists of member and member function declarations. The definition of lists and basic set theories are thought to be well-known.

Definition 1 *'B' is a set of system defined types such as int, float, char and etc.*

Definition 2 *'POINTER', 'ARRAY' and 'FUNCTION' are system defined symbols, and included in a special set 'S'.*

Definition 3 *'ST' is a set of strings.*

Definition 4 *For all list x, y, let 'pointer(x)' return '$\langle POINTER, x\rangle$', 'array(x)' return '$\langle ARRAY, x\rangle$', 'id(x)' return '$\langle x\rangle$', and 'function($\langle x,y\rangle$)' return $\langle FUNCTION,\ x,\ y\rangle$. Then, a set Ty is defined to be a set of functions generated by applying the association operator '$\circ$' repeatedly to the elements of a set $Q = \{pointer(\ ),\ array(\ ),\ function(\ ),\ id(\ )\}$.*

The association operator has some limitations – for example, pointers of function are not allowed - but currently, we ignore them.

Definition 5 *For a given list $l = \langle l_1, \dots\ l_m\rangle$ s.t $l_1 \notin S$, tz_l is a set of lists defined as followings :*

1. *$aTy(l) \in tz_l$, when $aTy \in Q$ and $\forall\ aTy \in (Q - \{function()\})$ or*
2. *aTy is a 'function()' and $aTy(l_1,\ l_2) \in tz_l$, when $l = tz_{l_1} \cup tz_{l_2}$.*

And, 'ty' is defined as the function from l to tz_l.

tz_l represents the set of types which are constructed using the types in l and the type constructors in Q.

Definition 6 (Set of interfaces) *Let id and member-name be elements of ST. 'I' is the set of all interfaces each of which is defined as (id, ILIST) where ILIST is $\langle a_1,\ a_2,\ ..\ a_n\rangle$ s.t. for all $i (1 \le i \le n)$, a_i = (ty(j), member-name) with $j = \langle b_1, ..\ b_m\rangle$ and $b_i (1 \le i \le m) \in I \cup B$.*

Thus *ILIST* is a list of declarations of members and member functions of types in *I*.

Definition 7 (Set of implementations) *Let id, member-name$_k$ and member-name$_m$ be in the set ST. 'M' is the set of all implementations each of which is defined as (id, PUBLIST, PRLIST) where PUBLIST = $\langle a_1,\ a_2,\ ..\ a_{n_{pub}}\rangle$ and PRLIST = $\langle b_1,\ b_2,\ ..\ b_{n_{pr}}\rangle$ for a_k = (ty(i), member-name$_k$) and b_m = (ty(j), member-name$_k$). In there, $i = \langle c_1, ..\ c_p\rangle$ and $j = \langle d_1, ..\ d_p\rangle$, and both c_z and d_w are in $I \cup M \cup B$ where $(1 \le k \le n_{pub}$ and $1 \le m \le n_{pr})$ and $(1 \le z \le p$ and $1 \le w \le q)$[3].*

Thus, a *PUBLIST* and a *PRLIST* are lists of declarations of members and member functions with types in *I* or *M*. A *PUBLIST* represents the public part of a class, while a *PRLIST* stands for the private part.

Hierarchies Defined by Users

The hierarchies of interfaces and implementations are constructed by explicit specification of inheritance by users. Binding of an interface and an implementation is also defined by users.

[3]In C++, there are some other keywords such as `virtual`, `inline`, `const`, and `typedef` which are not important in this paper.

Definition 8 *'IH', 'IM', and 'II' are defined as followings:*

1. *IH = { (i_k, i_j) | for i_k, $i_j \in I$, there is a user defined inheritance relationship $\langle i_k, i_j \rangle$ }*
2. *IM = { (m_k, m_j) | for m_k, $m_j \in M$, there is a user defined inheritance relationship $\langle m_k, m_j \rangle$ }*
3. *II = { (i_k, m_j) | for $i_k \in I$, $m_j \in M$, there is a user defined binding $\langle i_k, m_j \rangle$ }*

IH and *IM* represents the hierarchy of interfaces and implementations, respectively. *II* denotes the relationships between implementation classes and interfaces. All of them are specified by users explicitly.

Wellformedness of IH and IM

The user-defined hierarchies of interfaces or implementations are meaningful only when they have '*wellformedness*' property described in the following definitions.

Definition 9 *For two lists $\langle a_1, a_2, ...a_n \rangle$ and $\langle b_1, b_2, ..b_m \rangle$, 'strict sublist' relationship or 'strict_sublist(a, b)' denotes the case in which $n < m$ and, for all $a_i (i <= n)$, $a_i = b_i$.*

Definition 10 (Wellformedness of IH) *IH is wellformed if for all (i_1, i_2) in IH, strict_sublist(ILIST(i_1), ILIST(i_2)) is true.*

Definition 11 (Wellformedness of IM) *IM is wellformed if, for all $(m_1, m_2) \in$ IM,*

1. *(PUBLIST(m_1) = PUBLIST(m_2) or, strict_sublist(PUBLIST(m_1), PUBLIST(m_2))),*
2. *(PRLIST(m_1) = PRLIST(m_2) or, strict_sublist(PRLIST(m_1), PRLIST(m_2)) is true) and,*
3. *not both PUBLIST(m_1) = PUBLIST(m_2) and PRLIST(m_1) = PRLIST(m_2) are true.*

The wellformedness of *IM* and *IH* is automatically satisfied, because of the grammar of an interface declaration. From now on, *IH* and *IM* are considered to be wellformed by default, unless there are special mentions on it.

Extensions from IH and IM

IH and *IM* are extended to *IH** and *IM** with reflexivity and transitivity.

Definition 12 *Two extended relationships are defined as followings.*

- *For i and j in I, a relationship IH* is defined on (i,j) if (1) i is equal to j, or (2) (i,j) is in IH, or (3) there exists $k \in I$ s.t (i,k) and (k,j) are in IH*.*
- *For m and l in M, a relationship IM* is defined on (m,l) if (1) m is equal to l, or (2) (m,l) is in IM, or (3) if there exists $n \in M$ s.t. (m,n) and (n,l) are in IM*.*

Next, we will extend I and M with $-_I$ and $-_M$.

Definition 13 (Extended Relationship) Ipo $= \langle I^+, IH^{*+} \rangle$ *is defined by* $I^+ = I \cup \{ -_I, \top_I \}$ *and* $IH^{*+} = IH^* \cup \{ (-_I, i) \mid \forall i \in I^+ \}$ *s.t.* $-_I \notin I$. *Similarly,* Mpo $= \langle M^+, IM^{*+} \rangle$ *is defined by* $M^+ = M \cup \{-_M\}$ *and* $IM^{*+} = IM^* \cup \{ (-_M, i) \mid \forall i \in M^+ \}$ *s.t.* $-_M \notin M$[4].

Conformance of Interface and Implementation

The type space for SII in C++ is constructed by combination of interfaces in I and implementation classes in M. But, as explained above, I and M have independent hierarchies and all the elements in both I and M are be recognized as type units in the resulting type system in C++.

Definition 14 *For* Mpo *and for all m in M, the set of gate nodes for m, in other words 'gate(m)', exists, and $x \in$ gate(m) if and only if :*

1. *there is j s.t. $(j,x) \in$ II and*
2. *there is no y s.t there exists k and $(k, y) \in$ II and $(x,y) \in$ II.*

Each element of '*gate(m)*' is the smallest super class in one of the paths from (but not including) m to the root, which has explicit specification of its interface.

Definition 15 (Conformance) *For an interface i and an implementation m, we define $(i,m) \in$ cf and 'm conforms to i', if (1) $(i,m) \in$ II, or (2) $(i,j) \in$ IH and $(j,m) \in$ II, or (3) $(i, gate(m)) \in$ cf.*

Since the wellformedness of *II* is defined based on the type conformance it is a little more complex than that of *IH* or *IM*.

Definition 16 (Wellformedness of II) *II is wellformed, if, for all $(i,m) \in$ II s.t. ILIST(i) = $(a_1, a_2, ..., a_z)$ and PUBLIST(m) = $(b_1, b_2, ..., b_z)$, either following two conditions is satisfied : (1) for all k ($1 \leq k \leq w$), $a_k = b_k$, or (2) for all k ($1 \leq k \leq w$), and*

[4] We found that IH^{*+} and IM^{*+} are the partial order sets[5].

for $a_k = (ty_k(j_k), mn)$, $b_k = (ty_k(l_k), mn)$, *there exist a list of I, named* j_k, *and a list of* $I \cup M$, *named* l_k, *and* $(l_{k,x}, j_{k,x})$ *is in II, where* $l_{k,x} \in l_k = \langle l_{k,1}, \ldots l_{k,z_k} \rangle$, *and* $j_{k,x} \in j_k = \langle j_{k,1}, \ldots j_{k,z_k} \rangle$.

The second condition in definition 16 explains that the interfaces used to the types of members or member functions in an interface definition can be overridden by corresponding implementation classes. A '*wellformed gate()*' is a gate function which is defined by '*wellformed II*' instead of *II*. And wellformed conformance is defined using *wellformed gate()* and *wellformed II*[5]. We will name this 'wcf'.

The Algebra of Type Space

We are going to describe the resulting type space through the definitions and theorems above. The the type space corresponds to a system which has interfaces, implementations, ordinary classes and integrated classes in its domain and their relationships in its operator.

Definition 17 (The Proposed Model) TPO $= \langle\langle I \cup \{\top_I\} \times M \cup \{\top_M\} \rangle, \Sigma\rangle$ *is defined by the definition of* Σ, *where* $\langle \langle i_1, m_1\rangle \langle i_2, m_2\rangle \rangle \in \Sigma$ *if and only if the four conditions, such as (1)* $\langle i_1, m_1\rangle \in wcf$, *(2)* $\langle i_2, m_2\rangle \in wcf$, *(3)* $\langle i_1, i_2\rangle \in IH^{*+}$, *(4)* $\langle m_1, m_2\rangle \in IM^{*+}$ *are satisfied(where,* $i_1, i_2 \in (I \cup \{\top_I\})$ *and* $m_1, m_2 \in (M \cup \{\top_M\})$*).*

We showed that TPO preserves IH^{*+}, IM^{*+} in [5].

Another Approach

As seen earlier, the SII model in Martin et al.[12] produces side effects which introduce redundant or even wrong binding of interfaces and implementations, so users are open to misuse types. In the next definition, 'THP' represents this model.

Definition 18 THP $\langle I^+ \times M^+, \Upsilon\rangle$ *is defined where, for* $i_1, i_2 \in I^+$ *and* $m_1, m_2 \in M^+$, Υ *is defined as*

1. $\langle (i_1, -_M), (i_2, -_M)\rangle \in \Upsilon$ *when* $(i_1, i_2) \in IH$
2. $\langle (-_I, m_1), (-_I, m_2)\rangle \in \Upsilon$ *when* $(m_1, m_2) \in IM$
3. $\langle (i_1, -_M), (i_1, m_1)\rangle \in \Upsilon$ *when* $(i_1, m_1) \in II$
4. $\langle (i_1, m_1), (-_M, M)\rangle \in \Upsilon$ *and* $\langle (-_M, M), (i_1, m_1)\rangle \in \Upsilon$ *when* $(i_1, m_1) \in II$
5. $\langle (i_1, m_1), (i_3, m_3)\rangle \in \Upsilon$, *when* $\langle (i_1, m_1), (i_2, m_2)\rangle \in \Upsilon$ *and,* $\langle (i_2, m_2), (i_3, m_3)\rangle \in \Upsilon$

Υ stands for the resulting class hierarchy in [12]. Formula 3 describes the semantics that a resulting class(a pair of an interface and an implementation) should inherit from the class representing its interface. Formula 4 means that the resulting class is considered to be equal to the implementation class.

The next theorem explains that, in such a model bindings which are not intended by users can be generated. Due to the limited space, detailed proof is not provided here[5].

Theorem 1 *For* $(i_1, m_1), (i_2, m_2) \in I \times M$, $(m_1, m_2) \in IM$, $\langle (i_1, m_1), (i_2, m_2)\rangle \in \Upsilon$ *of* THP.

Proof) Omitted. □

Thus, whether or not i_1 *and* i_2 are related each other by IH^{*+}, if (m_1, m_2) is in IM^{*+}, the resulting pair (i_1, m_1) and (i_2, m_2) are related in Υ. It results from the fact that there is no distinction between an implementation and a pair of the implementation and its interface.

Now, the next theorem is on the properties of our model, TPO. Due to the limited space, detailed proof is not provided here[5].

Theorem 2 *For* $(i_1, m_1), (i_2, m_2) \in I \times M$, $(m_1, m_2) \in IM$, *if* $i_1 \neq i_2$ *and* i_1, i_2 *are not related to each other in* IH^{*+}, $\langle (i_1, m_1), (i_2, m_2)\rangle \notin \Sigma$ *of* TPO.

Proof) Omitted. □

From the theorem 2, we can see that the side-effects produced in Martin et al.[12] does not occur in our model.

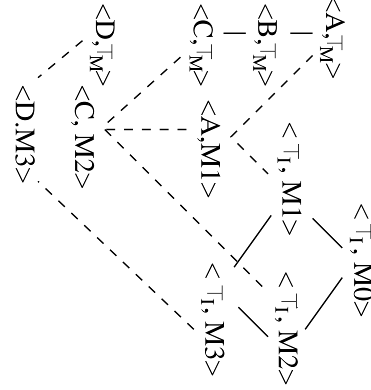


Figure 4: The type spaces of our model

3.2.2 Example

In our model, such an anomaly is solved, as illustrated in figure 4. The hierarchy in figure 4 is corresponding to that of Martin et. al[12] in figure 1. $\langle D, M3\rangle$ does not directly conform to $\langle C, \top_M\rangle$ because C and D are not related each other. However $\langle D, M3\rangle$ is still

	Tb	basic types
	Tbi	basic intf-types
	Tbm	basic impl-types
ti ∈	**Tyi** = **Tb** + **Tbi**	
	+ {`array`} × **N**× **Tyi**	
	+ {`Pointer`} × **Tyi**	intf-types
tm ∈	**Tym** = **Tb** + **Tbm**	
	+ {`array`} × **N** × **Tym**	
	+ {`Pointer`} × **Tym**	impl-types
t ∈	**Ty** = **Tyi** × **Tym** + **Tb**	types
ent ∈	**Ent** = **Ide** → [{`tag1`} × **Tyi**	
	+ {`tag2`} × **Tym** + **Ty**]	environments
acc ∈	**Acc** = {`private`, `public` }	access specifiers

Table 1: Semantic domains

related to $M0$, $M1$ and $M2$ because $M3$ is a subclass of those classes. Thus, the result relationships in our model are exactly what users originally intended to.

3.2.3 Denotational semantics

The formal model presented in the previous section may be used to analyze the concrete semantics of C++ with SII. In this section, we propose a denotational semantics based on the above algebraic definition.

The semantic domains are described in table 1. **Tb** corresponds to the basic types like `int` and `float` in C++, while **Tbi** and **Tbm** represents user defined interfaces and implementations(i.e. I and M) respectively. **Tyi** and **Tym** represents the types constructed from interfaces and from implementations, respectively. Both **Tyi** and **Tym** include **Tb**. A type can be described by one of **Tyi**, **Tym** or a pair of types from **Tyi** and **Tym**. **Ent** is an environment for symbols. In **Ent**, the symbols related to **Tyi** and **Tym** are differentiated by '`tag`'s from those which represent definitions of normal variables. The semantic rules for SII are presented in [5]

4 Comparisons

In table 2, we compare our model with two previous approaches by Martin et al.[12] and Baumgartner et al.[2]. The comparison criteria consists of (1) the number of implementation allowed to be bound to an interface, (2) existence of the concept of hierarchy of interface and implementation, (3) the way of subtyping, (4) the ways of binding of interfaces and implementations and (5) contents of an interface.

5 Conclusion

This paper introduces the mechanism for the separation of interface and implementation which extends C++, and makes the semantics of the separation of interface and implementation clearer with a formal model – algebraic definitions and denotational semantics. In this proposed model, the problems which occurred in the other existing models are solved.

The separation of interface and implementation in object-oriented languages becomes important more and more, especially for the applications such as database systems and distributed systems. Now, we are working on the other object oriented languages besides C++ in order to find some general properties of the separation of interface and implementation. The semantics of function calls and virtual base classes in C++ also needs further investigation.

References

[1] Francois Bancilhon, Claude Delobel, and Paris Kanellakis. *"Object-Oriented Database System - The Story of O_2"*. Morgan Kanfmann Publishers, Inc., 1991.

[2] G. Baumgartner and V. F. Russo. " Signatures: A C++ Extension for Type Abstraction and Subtype Polymorphism". Technical Report CSD-TR-93-059, Purdue University, September 1993.

[3] Peter S. Canning, William R. Cook, Walter L. Hill, and Walter G. Olthoff. "Interfaces for Strongly-Typed Object-Oriented Programming". In *Proc. of the ACM OOPSLA Conf.*, pages 457–467, October 1989.

[4] Giuseppe Castagna. "Covariance and Contravariance : Conflict without a Cause". *ACM Transactions on Programming Languages and Systems*, 17(3):431–447, 1995.

[5] E.S. Cho. "On the Extension of Strongly-Typed Object Oriented Language ". Technical report, Seoul National University, 1996.

[6] R.C.H Connor, A.L. Brown, Q.I Cutts, and A. Dearle. "Type Equivalence Checking in Persistent Object Systems". In *The Fourth International Workshop on Persistent Object Systems*, pages 154–167, 1990.

[7] William R. Cook. "Interfaces and Specifications for the Smalltalk-80 Collection Classes". In *Proc. of the ACM OOPSLA Conf.*, pages 1–15, 1992.

[8] David Gelernter and Suresh Jagannathan. *Programming Linguistics*. MIT Press, 1990.

Table 2: Comparison with the approaches to SII for C++

Items	In Martin et al.[12]	In Baumgartner et al.[2, 10]	Our Model
Implementaions/interface	more than one	more than one	more than one
Concept of hierarchy	for both interface and implementation	for both interface and implementation	for both interface and implementation
Subtyping	explicit specification	by structural equation	explicit specification
Binding of interface and implementation	explicit in class declarations	by strucutral equation	explicit in class declarations
Contents of interface	member function only	member function, etc.	member, member function

[9] A. Goldberg and D. Robson. *Smalltalk-80 : The Language and Its Implementation*. Addison-Wesley Publishing Company, Inc., 1983.

[10] Elana D. Granston and V. F. Russo. "Signature-Based Polymorphism for C++". In *Proceeding of Usenix C++ conference*, pages 65–79, 1991.

[11] Wilf R. LaLonde, Dave A. Thomas, and John R. Pugh. "An Exemplar Based Smalltalk". In *Proc. of the ACM OOPSLA Conf.*, 1986.

[12] Bruce Martin. "The Separation of Interface and Implementation in C++". In *Proceeding of Usenix C++ conference*, pages 51–63, 1991.

[13] R. K. Raj and et al. "The Emerald Approach to programming". Technical Report 88-11-01, University of Washington, November 1989.

[14] Craig Schaffert and et al. "An Introduction to Trellis/Owl". In *Proc. of the ACM OOPSLA Conf.*, pages 9–16, September 1986.

[15] P. Schwarz and et al. "Extensibility in the Startburst Database System". In *Proc. of the Conf. on VLDB*, pages 85–92, 1986.

[16] Bjarne Stroustrup, editor. *The C++ programming language second edition*. Addison-Wesley Publishing Company, Inc., April 1991.