Extension Rules: Description Rules for Safely Composable Aspects

Technical Report AIST01-J00002-4, Feb 2003

Yuuji ICHISUGI
National Institute of Advanced Industrial Science and Technology
AIST Tsukuba Central 2, Tsukuba,Ibaraki 305-8568, Japan
y-ichisugi@aist.go.jp

Akira TANAKA
National Institute of Advanced Industrial Science and Technology
AIST Tsukuba Central 2, Tsukuba,Ibaraki 305-8568, Japan
akr@m17n.org

Takuo WATANABE
National Institute of Informatics /
Tokyo Institute of Technology
2-1-2 Hitotsubashi, Chiyoda-ku, Tokyo 101-8430, Japan
takuo@acm.org

ABSTRACT

Aspect composition — application of two or more separately developed aspects to a single target code — generally requires great skills and knowledge on both aspects and the target. We give a collection of rules, called *extension rules*, for safely composable aspects. If all the member of a set of aspects obey the extension rules, they can be safely composable, even if independent programmers implement them. Our extension rules restrict how aspects extend the behavior of existing methods in the target. In this paper, we say how the extension rules can be applied to the diamond inheritance case of mixins and then *MixJuice* language modules. We describe a method of verifying extension rules in those cases. We use the notion of pre- and post-conditions and behavioral subtyping to define *the composability criterion* for mixins and MixJuice modules. Four specific extension rules (*the After Rule*, *the Plus Rule*, *the Functional Protocol Rule* and *the Disjoint Branch Rule*) and their verification method are presented. We also mention the future direction of our research needed to realize safe and easy aspect-oriented programming.

1. INTRODUCTION

The following questions are often asked about aspect-oriented programming languages.

- Isn't it dangerous to alter the behavior of existing classes?
- Is it possible to understand the source code without seeing the whole program?

This paper is a first step to answering these questions.

One important purpose of the modularization of programs is to enable the separate development of each module. When composing modules, developers hope that each module will work as well in concert with other modules as on its own. We call this property of modules *safely composable* or merely *composable*. On the other hand, when developing each module, ideally the programmer of the module can understand how the module works without needing to know the whole source code of modules to be composed. If this is possible, we say "*modular reasoning* is possible." These two properties of modules are actually the same thing but viewed from a different viewpoint.

In the case of a programming language that supports abstract data types (ADTs), rigorous *Design by Contract*[11] is required to make modules composable. (In this case, a module is an ADT definition.) In short, all programmers are required to obey the following rule: "The implementer of an ADT should implement the external interface of the ADT correctly, whereas the user of an ADT should not depend on the specific implementation of the ADT but depend only on the external interface of the ADT. " One way to represent the specifications of ADTs is as *pre- and post-conditions*. If programmers rigorously apply Design by Contract, the change of an internal implementation of an ADT does not change the correctness of other code that uses the ADT.

In the case of object-oriented languages with static type checking, another rule is required to make modules composable. (In this case, a module is a class definition.) The programmers of a subclass should guarantee that "A subclass instance is substitutable for a superclass instance." This rule can also be expressed as "When a subclass overrides the superclass method, the subclass may weaken the precondition and strengthen the post-condition of the method." If a subclass is defined in this way, we say that the subclass is *a behavioral subtype*[9] of the superclass. If programmers rigorously obey this rule, extensions of the behavior by a subclass will not change the correctness of other codes that use the superclass type, because the change in behavior does

not violate the external specification of the superclass.

In the case of object-oriented languages that support *mixins*, more rules are required to make modules composable. (In this case, a module is a class or a mixin definition; composition is multiple inheritance of mixins.) No previous research has formally discussed the rules needed to make mixins safely composable; however, empirical knowledge has been acquired. In this paper, we define a "mixin" as a class that is defined such that it will be multiply-inherited by other classes. If sets of method names defined by mixins are disjoint, these mixins are safely composable. On the other hand, it is usually dangerous for more than one mixin to override the same method. To support safe composition for the latter case, CLOS[16] and its predecessor, Flavors[12], provide a mechanism called *method combinations*. Nevertheless, the safety of method combinations has not been formally verified; in fact, they are not completely safe. Worse, there are no clear guidelines for designing new safe method combinations.

In this paper, we first propose a method for guaranteeing the safe composition of mixins by applying *extension rules*, which are generalizations of method combinations. We also explain how to formally verify extension rules. Extension rules restrict how mixins extend the behavior of superclass methods. If two mixins extend the behavior of superclass methods obeying an extension rule, two mixins are safely composable. In other words, extension rules guarantee that the behavior of a class that multiply-inherits mixins does not violate the external specification of each ancestor class.

Second, we apply our idea to MixJuice[6, 5], a kind of aspect-oriented language[4]. Making aspects safely composable has, up to now, been a problem remaining to be solved[4]. We believe our idea to be applicable to not only MixJuice, but also to other aspect-oriented languages, because some parts of the mechanisms of aspect-oriented languages inherently resembles mixins. For example, "before/after/around advice" of AspectJ[8] is originally a part of the method combination mechanism of Flavors. Our idea may increase the safety of this mechanism of AspectJ.

All the extension rules described in this paper have been verified in the case of diamond inheritance of mixins and MixJuice modules so far; however, we expect these rules to be safe even in cases of general inheritance graphs.

This paper is intentionally written in a descriptive and informal style. We assume that readers have a basic knowledge of object-oriented languages and first-order predicate logic.

We do not think that all programmers need to follow rigorous programming rules taking account of pre- and post-conditions to make aspects safely composable; instead, we believe it is possible to support adequately safe and easy aspect-oriented programming for programmers without knowing pre- and post-conditions, by enhancing the method combination mechanism of Flavors. In the final section, we cover the future direction of research needed to realize safe and easy aspect-oriented programming.

The rest of this paper is organized as follows. In Section 2, we briefly explain the notion of behavioral subtyping. In Section 3 and Section 4, extension rules for mixins and MixJuice modules are described respectively. In Section 5, we show a simple verification support tool. Section 6 covers related work. We conclude with Section 7.

2. BEHAVIORAL SUBTYPING

In this section, we briefly explain the notion of behavioral subtyping[9], which is used for our definition of composability.

Behavioral subtyping is a relationship that takes both signatures and behavior into consideration.

In many languages, such as C++ and Java, superclass declarations are combined with supertype declarations. For example, if a class C2 inherits another class C1, type C2 becomes a subtype of type C1. Because a subclass inherits all instance variables and methods from its superclass, all operations applicable to the superclass are also applicable to the subclass. In this sense, this subtype relation is safe.

This subtyping mechanism guarantees that the "method not found error" will never occur; however, it does not guarantee more than that. Needless to say, many programs do not work as expected even if the "method not found error" does not occur.

This problem surfaces if different programmers implement classes. For example, suppose that a method m of class A receives a parameter of type Vector.

```
class A {
  void m(Vector v){
    ...
    int s = v.size();
    ...
    Object x = v.elementAt(i);
    ...
  }
}
```

Usually, the implementer A of the class A assumes that the received parameter is an instance of the class Vector. If another programmer B defines a subclass of Vector whose behavior is completely different from Vector, and passes it to m, m does not work correctly. For example, an ArrayIndexOutOfBoundsException may occur.

To avoid such situations, a programming rule: "Instances of a subclass should be substitutable for instances of the superclass", is required[1]. That is to say, a programmer defining a new subclass is obliged to guarantee that instances of the subclass behave as instances of the superclass. If this rule is satisfied, the type of subclass is a *behavioral subtype* of the type of superclass.

The relation of behavioral subtyping can be defined more strictly by using logical formulas. The specification of a

[1]This requirement is also called the *Liskov Substitution Principle*[10].

method m is expressed as two logical formulas, a pre-condition and a post-condition. A pre-condition is expected to be satisfied before calling the method m. A post-condition is expected to be satisfied after returning from the method m. The following logical formulas need to show tautology[2] to make a class C2, whose pre- and post-conditions of method m are $R2$ and $E2$, a behavioral subtype of a class C1, whose pre- and post-conditions of method m are $R1$ and $E1$. (R and E mean "require" and "ensure", respectively.)

$$(R1 \Rightarrow R2) \wedge (R1 \Rightarrow (E2 \Rightarrow E1))$$

Note that class C2 needs to strengthen its post-condition only when the pre-condition $R1$ is satisfied[3].

For example, consider the following method m.

```
class C1 {
  double m(double p) { return Math.sqrt(p); }
}
```

This method receives a non-negative real number and returns its square root. The pre- and post-condition can be expressed as follows using parameter p and return value r of method m.

$$R1(p) \equiv p \geq 0$$
$$E1(p, r) \equiv r = \sqrt{p}$$

On the other hand, suppose that another class C2 has the following method.

```
class C2 {
  double m(double p) {
    if (p >= 0){
      return Math.sqrt(p);
    } else {
      return -1;
    }
  }
}
```

The specification of method m may be as follows.

$$R2(p) \equiv true$$
$$E2(p, r) \equiv (p \geq 0 \Rightarrow r = \sqrt{p}) \wedge (p < 0 \Rightarrow r = -1)$$

In this case, type C2 is a behavioral subtype of type C1 because the logical formula $(R1 \Rightarrow R2) \wedge (R1 \Rightarrow (E2 \Rightarrow E1))$, that is

$$(p \geq 0 \Rightarrow true)$$
$$\wedge\, (p \geq 0 \Rightarrow ((p \geq 0 \Rightarrow r = \sqrt{p}) \wedge (p < 0 \Rightarrow r = -1) \Rightarrow r = \sqrt{p}))$$

is tautological.

[2] As described in [9], other properties concerning *invariants* and *constraints* are required. The investigation of these properties is a future work.

[3] As pointed out in [3], the requirement $(R1 \Rightarrow R2) \wedge (E2 \Rightarrow E1)$ defined by [11] and [9] is too strong. In [3], a more weakened requirement $(R1 \Rightarrow R2) \wedge ((R2 \Rightarrow E2) \Rightarrow (R1 \Rightarrow E1))$ is used, which is equivalent to the requirement used in this paper.

Indeed, even if an instance of C2 is assigned to a variable of type C1, the return value of the method m is always equivalent to the value that C1 might return because the parameter passed to the method m is always a non-negative value.

Conversely, users of type C1 can program as if the values of variables of type C1 are always instances of C1. They need not be aware that the values may be instances of C2. That is to say, modular reasoning is possible.

We have assumed that the user of the type C1 rigorously applies Design by Contract. In other words, the user should not assume anything more than what is written in the external specification of C1. For example, the user should not expect the return value NaN (Not a Number) as a result of the method invocation of m with a negative parameter. Although the implementation of C1 happens to behave this way, this type of behavior is not included in the specification of C1.

Although it is not easy to apply Design by Contract and behavioral subtyping rigorously during actual programming, it is well known that they effectively enhance software reliability[11].

3. EXTENSION RULES FOR COMPOSABLE MIXINS

In this section, we consider, as a first step, diamond inheritance of mixins. (In the next section, we consider diamond inheritance of MixJuice modules by using notations and the verification method described in this section.)

3.1 Diamond Inheritance of Mixins

For consideration in this section, we will assume an object-oriented language with static typing, multiple inheritance and class linearization mechanisms.

Figure 1(a) is an example of diamond inheritance. Both class C2 and C3 inherit C1. Class C4 inherits both C2 and C3. We assume C4 itself defines no methods.

This language linearizes classes. The behavior of instances of class C4 is equivalent to the behavior of the instances of a class that is defined as shown in Figure 1(b). In this paper, we assume that no other results of linearization occur.

This language permits assignments of instances of subclasses to variables of type of superclasses.

A subclass may override its superclass's methods. The overriding method may invoke a superclass's overridden method by using a *super-invocation* language construct. Which method is actually invoked during runtime is determined by the result of linearization. For example, super invocation within class C3 in Figure 1(b) invokes the overridden method of class C2.

We assume that one method invocation of object O does not cause another method invocation of O. In other words, we do not examine the *downcall problem*[14].

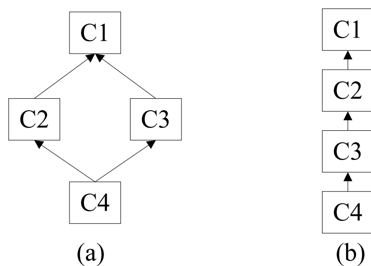


Figure 1: (a) The diamond inheritance of mixins C2 and C3. (b) The result of linearization of C4.

3.2 Notation for Behavior of Instances

We sometimes use a list of linearized classes to denote the behavior of an instance of a class because the precise behavior of an instance depends on the behavior of the super invocation within the instance, and the behavior of the super invocation is determined by the result of linearization. For example, the behavior of instances of classes C1, C2, C3 and C4 are denoted as $(C1)$, $(C1\ C2)$, $(C1\ C3)$ and $(C1\ C2\ C3\ C4)$ respectively.

The notation $o2 \hookrightarrow o1$ means the behavior of an instance $o2$ is a behavioral subtype of the behavior of an instance $o1$. For example, the following expression denotes that the instance of C2 is a behavioral subtype of the instance of C1.

$$(C1\ C2) \hookrightarrow (C1)$$

The above relation requires the following logical formula to be tautological for each method of class C1, as described in the previous section. ($R_{(C1)}$, $E_{(C1)}$, $R_{(C1C2)}$ and $E_{(C1C2)}$ are the pre- and post-conditions of $(C1)$ and the pre- and post-conditions of $(C1\ C2)$ respectively.)

$$(R_{(C1)} \Rightarrow R_{(C1C2)}) \wedge (R_{(C1)} \Rightarrow (E_{(C1C2)} \Rightarrow E_{(C1)}))$$

3.3 The Composability Criterion

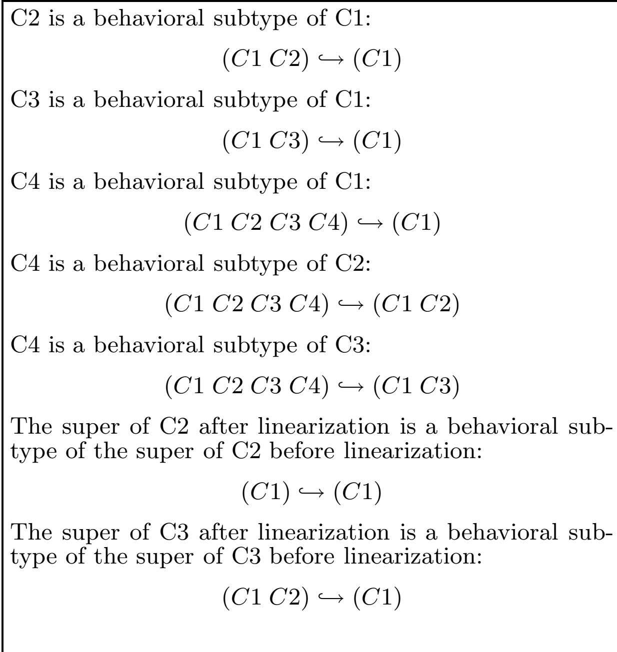
In this section, we define a criterion that judges whether C2 and C3 are safely composable or not.

This criterion consists of two sets of requirements. The first requires a behavioral subtyping relationship between superclasses and subclasses. The second requires a behavioral subtyping relationship between the behavior of the super invocations before and after linearization.

For example, in the first set of requirements, the instance of C4 needs to be a behavioral subtype of the instance of C3. This requirement is expressed as follows.

$$(C1\ C2\ C3\ C4) \hookrightarrow (C1\ C3)$$

Another set of requirements is necessary for modular reasoning concerning super invocations. For example, the properties of the super invocation expected by the programmer of C3 should be satisfied even if C3 becomes one of the linearized classes of C4. The programmer of C3 assumes the behavior of the super to be $(C1)$ as shown in Figure 1(a). On the other hand, the behavior of the super of C3 within


C2 is a behavioral subtype of C1:

$$(C1\ C2) \hookrightarrow (C1)$$

C3 is a behavioral subtype of C1:

$$(C1\ C3) \hookrightarrow (C1)$$

C4 is a behavioral subtype of C1:

$$(C1\ C2\ C3\ C4) \hookrightarrow (C1)$$

C4 is a behavioral subtype of C2:

$$(C1\ C2\ C3\ C4) \hookrightarrow (C1\ C2)$$

C4 is a behavioral subtype of C3:

$$(C1\ C2\ C3\ C4) \hookrightarrow (C1\ C3)$$

The super of C2 after linearization is a behavioral subtype of the super of C2 before linearization:

$$(C1) \hookrightarrow (C1)$$

The super of C3 after linearization is a behavioral subtype of the super of C3 before linearization:

$$(C1\ C2) \hookrightarrow (C1)$$

Figure 2: The composability criterion.

instances of C4 is $(C1\ C2)$ as shown in Figure 1(b). Therefore, this requirement is expressed as follows.

$$(C1\ C2) \hookrightarrow (C1)$$

All of these two kinds of requirements are listed in Figure 2. *The composability criterion* requires that all of these requirements be satisfied. If different programmers implement C2 and C3 without any restrictions, we cannot expect these requirements to be satisfied. Extension rules guarantee the composability criterion to be satisfied by restricting the behavior specifications of C2 and C3. If the composability criterion is satisfied, C4 can safely inherit C2 and C3 simultaneously. In other words, all assumptions about method invocations at the coding-time of each class are guaranteed to hold even after linearization.

3.4 The Representation of Specifications of Methods

In this section, we explain the representation of specifications of methods. (In the next section, we explain extension rules and their validity using this representation.)

Class C1, C2 and C3 introduce internal states $s1$, $s2$ and $s3$ respectively. (The internal state of the instance of C2 is a direct product of $s1$ and $s2$.)

The pre-condition of each method of class C1 is expressed as a logical formula containing state $s1$ before the invocation and parameter p. The post-condition of each method is expressed as a logical formula containing state $s1$ before the invocation, state $s1$' after the invocation, parameter p and return value r. The following is an example of precondition $R1$ and post-condition $E1$ of a method m and an implementation of the specification.

$R1(s1, p) \equiv true$
$E1(s1, s1', p, r) \equiv (s1' = p) \wedge (r = p)$

```
class C1 {
  int s1;
  int m(int p) { s1 = p; return p; }
}
```

When representing the specifications of C2 and C3, linearization should be taken into account, since the precise behaviors of instances depend on the results of linearization. More accurately, linearization affects the behavior of the super invocation. We use RO, EO and sO to denote the pre- and post-conditions and the state of the super respectively when writing the specifications of C2 and C3.

$RO(s1, sO, p)$
$EO(s1, s1', sO, sO', p, r)$

The state sO is introduced by other mixins that are inserted before the current class and the class C1 as a result of linearization. For example, sO seen from C2 within an instance $(C1\ C2)$ is empty; sO seen from C3 within an instance $(C1\ C2\ C3)$ is $s2$.

The following is an example of specification and implementation of C2 and C3. In this example, C2 invokes super and C3 does not.

$R2(s1, sO, s2, p) \equiv RO(s1, sO, p)$
$E2(s1, s1', sO, sO', s2, s2', p, r) \equiv$
$\quad EO(s1, s1', sO, sO', p, r) \wedge (s2' = p)$

```
class C2 extends C1 {
  int s2;
  int m(int p){ s2 = p; return super.m(p); }
}
```

$R3(s1, sO, s3, p) \equiv true$
$E3(s1, s1', sO, sO', s3, s3', p, r) \equiv$
$\quad (s1' = p) \wedge (sO' = sO) \wedge (s3' = s3) \wedge (r = p)$

```
class C3 extends C1 {
  int m(int p){ s1 = p; return p; }
}
```

After linearization, the behavior of each instance can be expressed without using RO, EO or sO, because the behavior of the super is determined. In general, the behavior of an instance $(C_1\ \cdots\ C_n\ C_{n+1})$ is obtained by substituting the behavior of an instance $(C_1\ \cdots\ C_n)$ for RO and EO of the specification of class C_{n+1}. The behavior of an instance (C_1) is the behavior of class C_1.

For example, the post-condition of a method m of an instance of C2, $E_{(C1C2)}$ is obtained as follows, by substituting the post-condition of class C1 for EO in the post-condition of class C2.

$E_{(C1C2)}(s1, s1', s2, s2', p, r) \equiv$
$\quad (s1' = p) \wedge (r = p) \wedge (s2' = p)$

The post-condition of a method m of an instance of C4, $E_{(C1C2C3C4)}$, is as follows. (Because C4 itself does not have any method definitions, it inherits C3's methods. The state sO in the specification of C3 is substituted by $s2$.)

$E_{(C1C2C3C4)}(s1, s1', s2, s2', s3, s3', p, r) \equiv$
$\quad (s1' = p) \wedge (s2' = s2) \wedge (s3' = s3) \wedge (r = p)$

The set of specifications of C1, C2, C3 and C4 described in this section is an example that causes a problem of multiple inheritance. Because C3 does not invoke super, the C2's method that should update $s2$'s value is not invoked. In fact, these specifications do not satisfy the composability criterion. One of the requirements listed in Figure 2, $(C1\ C2\ C3\ C4) \hookrightarrow (C1\ C2)$, is not satisfied, since $E_{(C1C2C3C4)} \Rightarrow E_{(C1C2)}$ is not tautological.

3.5 Extension Rules and Their Verification

In this section, we describe extension rules that restrict the specification of each class. We also explain the validity of the extension rules based on the composability criterion.

3.5.1 Roles of Framework and Extension Modules

We assume that C1 plays the role of a framework and that C2 and C3 act as extension modules.

There are probably an infinite number of extension rules that satisfy the composability criterion. The programmer who defines C1 should declare one extension rule for each method defined at C1. (It will be declared by a language construct provided by the programming language in the future, or by documents written in a natural language.) Both C2 and C3 override each method according to the extension rule declared at C1.

This style follows the programming style of Flavors using method combinations.

3.5.2 The After Rule

In this section, we describe one of the specific extension rules that satisfy the composability criterion.

We have designed the extension rule, which we'll call *the After Rule*, based on our programming experience. This rule is similar to the "after daemon", part of Flavors' method combination mechanism.

This rule compels the methods of the linearized classes to be executed from the top to the bottom of the linearized list and requires that each method preserves the super's post-condition.

The After Rule:
When a mixin overrides the superclass's method,

- it must not change the pre-condition of the method.
- it must invoke super exactly once at the beginning of the method execution.
- it must pass the parameters of the method to the super without changing them.
- it must return the super's return value without changing it.
- it may refer to the inherited state but must not update it.
- it may refer to and update the state introduced by itself.

This After Rule can be formally written as follows:

The After Rule:
The pre- and post-conditions of the method m of each mixin Ci (i = 2,3) that extends C1's m should be written in the following form.

$$R_i(\cdots) \equiv RO(s1, sO, p)$$
$$E_i(\cdots) \equiv EO(s1, s1', sO, sO', p, r) \wedge E_i'(s1', si, si', p, r)$$

E_i' is an arbitrary logical formula that contains $s1', si, si', p$ and r.

It is easy to confirm that the After Rule enforces the composability criterion on C2 and C3. For example, one of the requirements of the composability criterion $(C1\ C2\ C3\ C4) \hookrightarrow (C1\ C2)$, is satisfied as follows.

The pre- and post-conditions of an instance $(C1\ C2\ C3\ C4)$ are as follows.

$$R_{(C1\ C2\ C3\ C4)} \equiv R1$$
$$E_{(C1\ C2\ C3\ C4)} \equiv E1 \wedge E2' \wedge E3'$$

The pre- and post-conditions of an instance $(C1\ C2)$ are as follows.

$$R_{(C1\ C2)} \equiv R1$$
$$E_{(C1\ C2)} \equiv E1 \wedge E2'$$

In this case, the requirement $(C1\ C2\ C3\ C4) \hookrightarrow (C1\ C2)$, which is the following logical formula, is clearly tautological.

$$\begin{aligned}&(R_{(C1\ C2)} \Rightarrow R_{(C1\ C2\ C3\ C4)}) \\ &\quad \wedge (R_{(C1\ C2)} \Rightarrow (E_{(C1\ C2\ C3\ C4)} \Rightarrow E_{(C1\ C2)})) \\ &= (R1 \Rightarrow R1) \wedge (R1 \Rightarrow (E1 \wedge E2' \wedge E3' \Rightarrow E1 \wedge E2')) \\ &= \mathit{true}\end{aligned}$$

3.5.3 The Plus Rule

In this section, another example of an extension rule, *the Plus Rule*, is explained. This rule is similar to the "+" method combination of Flavors.

The Plus Rule is almost the same as the After Rule; methods of the linearized classes are executed from the top to the bottom; however, they differ in the following ways. Although the After Rule prohibits changing the return value of the super, the Plus Rule allows mixins to return a value that is the sum of the return value of the super and an arbitrary non-negative value. In compensation for this, each method cannot guarantee an exact return value; it only guarantees the minimum value of the return value.

The following program is an example of obeying the plus-rule.

```
class C1 {
  Vector v1;
  int m(){ return v1.size(); }
}
class C2 extends C1 {
  Vector v2;
  int m(){ return super.m() + v2.size(); }
}
class C3 extends C1 {
  Vector v3;
  int m(){ return super.m() + v3.size(); }
}
```

The Plus Rule can be formally written as follows:

The Plus Rule:
The post-condition of a method m of the class C1 and the pre- and post-conditions of m of each mixin Ci (i = 2,3) that extends C1 should be written in the following form.

$$E1(\cdots) \equiv \exists r1\ .\ E1'(s1, s1', p, r1) \wedge (r \geq r1)$$
$$R_i(\cdots) \equiv RO(s1, sO, p)$$
$$E_i(\cdots) \equiv \exists rO, ri\ .\ EO(\cdots, rO) \wedge Ei'(s1', si, si', p, ri) \wedge (ri \geq 0) \wedge (r \geq rO + ri)$$

It is easy to confirm that the Plus Rule enforces the composability criterion on C2 and C3. For example, one of the requirements of the composability criterion, $(C1\ C2\ C3\ C4) \hookrightarrow (C1\ C2)$, is satisfied as follows.

The pre- and post-conditions of an instance $(C1\ C2\ C3\ C4)$ are as follows.

$$R_{(C1\ C2\ C3\ C4)} \equiv R1$$
$$\begin{aligned}E_{(C1\ C2\ C3\ C4)} \equiv\ &\exists r1, r2, r3\ .\ E1' \wedge E2' \wedge E3' \\ &\wedge (r2 \geq 0) \wedge (r3 \geq 0) \wedge (r \geq r1 + r2 + r3)\end{aligned}$$

The pre- and post-conditions of an instance $(C1\ C2)$ are as follows.

$$R_{(C1\ C2)} \equiv R1$$
$$E_{(C1\ C2)} \equiv \exists r1, r2\ .\ E1' \wedge E2' \wedge (r2 \geq 0) \wedge (r \geq r1 + r2)$$

In this case, the requirement $(C1\ C2\ C3\ C4) \hookrightarrow (C1\ C2)$, which is the following logical formula, is clearly tautological.

$$\begin{aligned}&(R_{(C1\ C2)} \Rightarrow R_{(C1\ C2\ C3\ C4)}) \\ &\quad \wedge (R_{(C1\ C2)} \Rightarrow (E_{(C1\ C2\ C3\ C4)} \Rightarrow E_{(C1\ C2)})) \\ &= (R1 \Rightarrow R1) \wedge \\ &\quad (R1 \Rightarrow (\exists r1, r2, r3\ .\ E1' \wedge E2' \wedge E3' \wedge (r2 \geq 0) \wedge (r3 \geq 0) \\ &\qquad \wedge (r \geq r1 + r2 + r3) \\ &\qquad \Rightarrow \exists r1, r2\ .\ E1' \wedge E2' \wedge (r2 \geq 0) \wedge (r \geq r1 + r2))) \\ &= \mathit{true}\end{aligned}$$

We can easily think up variations of the Plus Rule. In general, extension rules that monotonically increase (or decrease) the return value of the super in certain meanings will satisfy the composability criterion. For example, an extension rule that permits the addition of elements to a set will satisfy the composability criterion. More practical examples include the rule for methods adding entries to hash tables and the rule for initialization methods of GUI menu entries.

The Plus Rule is an example of a "domain-specific" extension rule. This rule uses predicates of number theory, and its proof is based on the following theorem in number theory.

$$(a \geq b + c) \wedge (c \geq 0) \Rightarrow (a \geq b)$$

Likewise, actual programming of practical applications will require extension rules that use predicates of their domain. For example, an extensible compiler framework will require extension rules that use predicates of language semantics.

3.5.4 The Functional Protocol Rule

We have found an extension rule that permits changing the algorithm of calculation of the return value. We call this rule *the Functional Protocol Rule* because these kinds of methods are classified with functional protocols in [7][4].

Although all extension rules described so far force super invocations, the Functional Protocol Rule does not.

The following program is an example of obeying this rule. Two mixins independently cache the return value with a different algorithm.

```
class C1 {
  String m(String s){ ...; return r; }
}
class C2 extends C1 {
  Hashtable cache = new Hashtable();
  String m(String s){
    String r = (String)cache.get(s);
    if (r == null){
      r = super.m(s);
      cache.put(s, r);
    }
    return r;
  }
}
class C3 extends C1 {
  String lastS = null;
  String lastR = null;
  String m(String s){
    if (! (lastS != null && s.equals(lastS))){
      lastS = s; lastR = super.m(s);
    }
    return lastR;
  }
}
```

The Functional Protocol Rule can be formally written by expressing the fact that "side effects of methods may or may not occur." $E1'$ is a logical formula for determining the return value. $E1''$ and E_i'' are logical formulas for determining side effects. The side effects of C1 may or may not occur. C2 and C3 may or may not invoke super; their side effects may or may not occur. In any case, $E1'$ determines the return value.

[4] To be precise, an overrider of a method following a functional protocol in [7] is assumed to change not only the algorithm, but also the value to be returned. This kind of change is safe if done by one subclass; however, it is unsafe if done by mixins. Therefore, the extension rule described in this section prohibits changing the return value of the method.

> The Functional Protocol Rule:
> The post-condition of a method m of the class C1 and the pre- and post-conditions of m of each mixin Ci (i = 2,3) that extends C1 should be written in the following form.
>
> $$E1(s1, s1', p, r) \equiv (s1 = s1' \vee E1''(s1, s1', p)) \wedge E1'(s1, p, r)$$
> $$R_i(s1, sO, si, p) \equiv RO(s1, sO, p)$$
> $$E_i(s1, s1', sO, sO', si, si', p, r) \equiv (EO(s1, s1', sO, sO', p, r) \vee (s1 = s1' \wedge sO = sO' \wedge E1'(s1, p, r))) \wedge (si = si' \vee E_i''(s1', si, si', p))$$

3.5.5 The Disjoint Branch Rule

Another extension rule, which we call *the Disjoint Branch Rule*, permits extension of domain of the method. This extension rule assumes that all domains processed by mixins are disjoint.

The following program is an example of obeying this rule.

```
class C1 {
  void m(String s){}
}
class C2 extends C1 {
  void m(String s){
    if (s.equals("A")){...}
    else { super.m(s); }
  }
}
class C3 extends C1 {
  void m(String s){
    if (s.equals("B")){...}
    else { super.m(s); }
  }
}
```

One of the possible applications of this extension rule is a method that processes XML. If each tag processed by each mixin belongs to the name-space managed by the programmer of the mixin, the disjointness of each domain is guaranteed.

The Disjoint Branch Rule can be formally written as follows:

> The Disjoint Branch Rule:
> The pre- and post-conditions of C1's method m and the method m of each mixin Ci (i = 2,3) that extends C1's m should be written in the following form. (Each e_i is a constant, $\forall i, j (i \neq j) \, . \, e_i \neq e_j$.)
>
> $$R1(s1, p) \equiv false$$
> $$E1(s1, s1', p, r) \equiv true$$
> $$R_i(s1, sO, si, p) \equiv RO(s1, sO, p) \vee p = e_i$$
> $$E_i(s1, s1', sO, sO', si, si', p, r) \equiv ((p = e_i) \Rightarrow E_i'(s1, s1', si, si', p, r)) \wedge ((p \neq e_i) \Rightarrow EO(s1, s1', sO, sO', p, r))$$

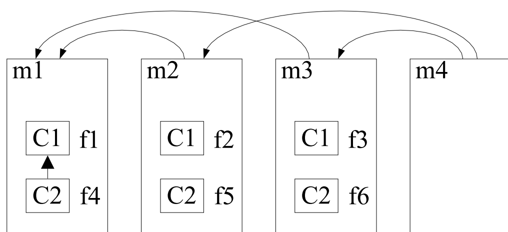


Figure 3: The diamond inheritance of MixJuice modules.

```
module m1 {
 define class C1 { define int m(int p){...}}    // f1
 define class C2 extends C1 { int m(int p){...}}// f4
}
module m2 extends m1 {
 class C1 { int m(int p){...}}  // f2
 class C2 { int m(int p){...}}  // f5
}
module m3 extends m1 {
 class C1 { int m(int p){...}}  // f3
 class C2 { int m(int p){...}}  // f6
}
module m4 extends m2, m3 {}
```

Figure 4: Definitions of MixJuice modules.

4. EXTENSION RULES FOR COMPOSABLE MIXJUICE MODULES

In this section, we describe extension rules for composable MixJuice modules using the representation and the verification method described in the previous section.

4.1 Overview of MixJuice Language

MixJuice[6, 5] is an object-oriented language that adopts a difference-based module mechanism instead of the class-based module mechanism of Java language.

Modules may inherit other modules. In MixJuice, both the module-inheritance mechanism and the traditional class-inheritance mechanism can be used independently. Class inheritance and module inheritance are different, as described next. Class inheritance is a mechanism for describing the difference between classes. Module Inheritance is a mechanism for describing the difference between two programs consisting of zero or more classes. Using class inheritance, the programmers can only define a new class that has a different name from that of the original class. By module inheritance, the programmers can modify the definitions of existing classes and methods without changing their names. Class inheritance is a mechanism for subtyping and safe late binding. Module inheritance is a mechanism for static reuse and information hiding.

Each module can be separately compiled. Although each module contains fragments of classes, such fragments are type-checked by the compiler. When compiling a module, the compiler requires only the ancestor modules of the module. Therefore, sibling modules can be developed by different programmers. End-users can compose independently developed modules, without requiring detailed knowledge of implementation of the modules.

Our aim is to guarantee the safety of the composition done by not only programmers, but also end-users.

4.2 Diamond Inheritance of MixJuice Modules

We now consider diamond inheritance of MixJuice modules, as in Figure 3. Two modules, m2 and m3, inherit the module m1, which have two class definitions of C1 and C2. The module m4 inherits both m3 and m4. Figure 4 is a MixJuice program corresponding to Figure 3.

An executable program is constructed by linking MixJuice modules. Before linking, the MixJuice linker linearizes modules in a certain algorithm. We express a linked program as a linearized list of module names. Programs produced from possible combinations of the four modules in Figure 3 are listed below. (We assume that no other results of linearization occur.)

$(m1)$
$(m1\ m2)$
$(m1\ m3)$
$(m1\ m2\ m3\ m4)$

In Figure 3 and Figure 4, $f1$, $f2$, ... , $f6$ mean "class fragments." Fragments $f1$ and $f4$ are class definitions of C1 and C2, respectively. Fragments $f2$ and $f3$ are extensions of class C1; $f5$ and $f6$ are extensions of class C2.

The behavior of an instance of each class in a linked program is denoted as a list of fragments. For example, in the program $(m1\ m2)$, the behavior of instances of C1 and C2 are $(f1\ f2)$ and $(f1\ f2\ f4\ f5)$ respectively; in the program $(m1\ m2\ m3\ m4)$, the behavior of instances of C1 and C2 are $(f1\ f2\ f3)$ and $(f1\ f2\ f3\ f4\ f5\ f6)$ respectively.

4.3 The Composability Criterion

In this section, we describe the composability criterion for diamond inheritance as shown in Figure 3.

The basic approach is the same as in Section 3. The composability criterion consists of two kinds of requirements. The first kind of requirement is for modular reasoning concerning method invocations for objects, and the second kind of requirement is for modular reasoning concerning super invocations.

Determining the first kind of requirement is more complex than in the case of mixins because both class-inheritance and module-inheritance mechanisms are used. For example, the programmer of the module m2 assumes that the behavior of the instance is assigned to the variable of type C2 is $(f1\ \ f2\ \ f4\ \ f5)$. Nevertheless, when the program $(m1\ m2\ m3\ m4)$ is executed, the behavior of the instance of C2 is $(f1\ \ f2\ \ f3\ \ f4\ \ f5\ \ f6)$. Therefore, the following requirements should be satisfied.

$$(f1\ f2\ f3\ f4\ f5\ f6) \hookrightarrow (f1\ f2\ f4\ f5)$$

Determining another kind of requirement is also more complex. In MixJuice, there are two kinds of super invocations. One invokes the method overridden by class-inheritance, and the other invokes the method overridden by module-inheritance. These two kinds of super invocations are actually the same mechanism, since both of them invoke a method of a previous class fragment in the list of linearized fragments[5]. For example, in the program $(m1\ m2)$, the super invocation from $f5$ invokes $f4$, not $f2$.

Taking these into account, we have listed the requirements of the composability criterion in Figure 5. $C_j@m_i$ is an abbreviation for the behavior of the instance of the class C_j in the module m_i. $s_j@m_i$ is an abbreviation for the behavior of the super seen from f_j in the module m_i.

Note that the following properties of lists of linearized fragments hold.

- When $f2$ or $f3$ is in a list, $f1$ is present to the left of them.
- When $f5$ or $f6$ is in a list, $f4$ is present to the left of them and $f1$ is present to the left of $f4$.

These properties are caused by the MixJuice language specification. In MixJuice, each class needs to be defined at a module m and may be extended at other modules that extend m. The super class of the class is declared at the module m and cannot be changed by the other modules.

Because of these properties, these class fragments are not necessarily symmetric. For example, although $f5$ and $f6$ are symmetric, $f4$ and $f5$ are not. This asymmetry is reflected in the extension rule described in the next section.

4.4 The MixJuice After Rule

We have defined the *MixJuice After Rule* listed in Figure 6 by modifying the After Rule for mixins. One example of "method extension by subclass" is method overriding done by $f4$. Examples of "method extension by sub-module" are method overriding done by $f2$, $f3$, $f5$ and $f6$.

Please note the following:

Note 1: In traditional object-oriented languages, subclasses do not have to invoke super as long as subclasses obey the rule, "The pre-condition may be weakened and the post-condition may be strengthened." On the other hand, in the case of MixJuice languages, super invocation is mandatory because sub-module may extend the behavior of the super-class. For example, if $f4$ does not invoke super, the post-condition of super extended by $f2$ and $f3$ may not be satisfied, because the programmer of $f4$ does not know the specifications of $f2$ and $f3$.

Note 2: Access rules for the inherited state in the MixJuice After Rule are different from those of the Mixin After Rule. In the case of the Mixin After Rule, the inherited state is allowed to be referred to but not to be updated. On the other hand, we have designed the MixJuice After Rule so that subclasses may refer to and/or update the inherited state and sub-modules must neither refer to nor update the inherited state. (Although there may be other possible design choices, we selected this choice because we want to create freedom of extension by subclass in MixJuice as in traditional object-oriented languages.)

If the rule "The sub-module must not refer to the inherited state" were absent, the following problem will occur. Assume that the post-condition of a method m of each f_i in modules m1 and m2 is as follows.

- The post-condition of $f1$: $(s1' > 0)$
- The post-condition of $f2$: $(s2' = s1')$ (Corresponding to the execution of an assignment: "`s2 = s1;`")
- The post-condition of $f4$: $(s1' = 1)$ (Corresponding to the execution of an assignment: "`s1 = 1;`")

In this case, if the implementation of $f1$ happen to be "`s1 = 2;`", the program $(m1\ m2)$ may not work correctly. Although the module m2 is implemented on the assumption that the result of the method invocation of C2 is $(s2' = s1')$, the actual result of the invocation is $(s1' = 1)$ and $(s2' = 2)$.

Nevertheless, if the exact value of the inherited state s1' is predictable from the post-condition of the super-module, the sub-module *may* refer to it. For example, if the post-condition of $f1$ is $(s1' = 1)$, $f2$ and $f3$ may refer to s1' because it is actually equivalent to the constant value 1.

Note 3: If $f2$ refers to the return value of the super, e.g. "`s2 = super.m();`", and $f4$ changes the return value, a problem may occur. However, such references are allowed if the exact return value is predictable from the post-condition of the super-module, as well as Note 2.

The MixJuice After Rule can be formally written as follows:

The MixJuice After Rule:
1. When a subclass C2 extends a method m of a superclass C1, the pre- and post-conditions of the method m of C2 should be written in the following form.

$$R2(s1, sO, s2, p) \equiv RO(s1, sO, p) \vee R2'(s1, s2, p)$$

$$\begin{aligned} E2(\ldots, r) \equiv\ & (RO(s1, sO, p) \Rightarrow \exists s1'', r''\ . \\ & \qquad (EO(s1, s1'', sO, sO', p, r'') \\ & \qquad \wedge E2'(s1'', s1', s2, s2', p, r'', r) \\ & \qquad \wedge E1(s1, s1', p, r))) \\ & \wedge (\neg RO(s1, sO, p) \Rightarrow E2''(s1, s1', s2, s2', p, r)) \end{aligned}$$

2. When a sub-module m_i (i = 2,3) extends a method m of a super-module m1, the pre- and post-conditions of the method m of m_i should be written in the following form.

$$R_i(s1, sO, s_i, p) \equiv RO(s1, sO, p)$$

$$E_i(s1, s1', sO, sO', si, si', p, r) \equiv EO(s1, s1', sO, sO', p, r) \wedge E_i'(s_i, s_i', p)$$

[5] Actual MixJuice language uses *original-invocation* language constructs for both kinds of invocations, instead of Java's super-invocation language construct.

Requirements for the program $(m1)$:

$C2@m1 \hookrightarrow C1@m1$, that is: $(f1\ f4) \hookrightarrow (f1)$

Requirements for the program $(m1\ m2)$:

$C1@m2 \hookrightarrow C1@m1$, that is: $(f1\ f2) \hookrightarrow (f1)$

$C2@m2 \hookrightarrow C1@m1$, that is: $(f1\ f2\ f4\ f5) \hookrightarrow (f1)$
$C2@m2 \hookrightarrow C1@m2$, that is: $(f1\ f2\ f4\ f5) \hookrightarrow (f1\ f2)$
$C2@m2 \hookrightarrow C2@m1$, that is: $(f1\ f2\ f4\ f5) \hookrightarrow (f1\ f2\ f4)$

$s4@m2 \hookrightarrow s4@m1$, that is: $(f1\ f2\ f4) \hookrightarrow (f1)$

Requirements for the program $(m1\ m3)$:

Similar to the requirements for $(m1\ m2)$.

Requirements for the program $(m1\ m2\ m3\ m4)$:

$C1@m4 \hookrightarrow C1@m1$, that is: $(f1\ f2\ f3) \hookrightarrow (f1)$
$C1@m4 \hookrightarrow C1@m2$, that is: $(f1\ f2\ f3) \hookrightarrow (f1\ f2)$
$C1@m4 \hookrightarrow C1@m3$, that is: $(f1\ f2\ f3) \hookrightarrow (f1\ f3)$

$C2@m4 \hookrightarrow C1@m1$, that is: $(f1\ f2\ f3\ f4\ f5\ f6) \hookrightarrow (f1)$
$C2@m4 \hookrightarrow C1@m2$, that is: $(f1\ f2\ f3\ f4\ f5\ f6) \hookrightarrow (f1\ f2)$
$C2@m4 \hookrightarrow C1@m3$, that is: $(f1\ f2\ f3\ f4\ f5\ f6) \hookrightarrow (f1\ f3)$
$C2@m4 \hookrightarrow C2@m1$, that is: $(f1\ f2\ f3\ f4\ f5\ f6) \hookrightarrow (f1\ f4)$
$C2@m4 \hookrightarrow C2@m2$, that is: $(f1\ f2\ f3\ f4\ f5\ f6) \hookrightarrow (f1\ f2\ f4\ f5)$
$C2@m4 \hookrightarrow C2@m3$, that is: $(f1\ f2\ f3\ f4\ f5\ f6) \hookrightarrow (f1\ f3\ f4\ f6)$

$s2@m4 \hookrightarrow s2@m2$, that is: $(f1) \hookrightarrow (f1)$
$s3@m4 \hookrightarrow s3@m3$, that is: $(f1\ f2) \hookrightarrow (f1)$
$s4@m4 \hookrightarrow s4@m1$, that is: $(f1\ f2\ f3) \hookrightarrow (f1)$
$s5@m4 \hookrightarrow s5@m2$, that is: $(f1\ f2\ f3\ f4) \hookrightarrow (f1\ f2\ f4)$
$s6@m4 \hookrightarrow s6@m3$, that is: $(f1\ f2\ f3\ f4\ f5) \hookrightarrow (f1\ f3\ f4)$

Figure 5: The composability criterion for the diamond inheritance of MixJuice modules.

The MixJuice After Rule:

1. In the case of method extension by subclass:
    - It may weaken the pre-condition of the method.
    - If the received parameters satisfy the pre-condition of the super,
        - it may strengthen the post-condition of the method.
        - it must invoke super exactly once at the beginning of the method execution. (Note 1)
        - it must pass the parameters of the method to the super without changing them.
        - it may return a value different from the super's return value.
        - it may refer to the inherited state.
        - it may update the inherited state. (Note 2)
        - it may refer to and update the state introduced by itself.
    - If the received parameters do not satisfy the pre-condition of the super,
        - there are no restrictions on the post-condition of the method.
        - it must not invoke super.
2. In the case of method extension by sub-module:
    - It must not change the pre-condition of the method.
    - It must invoke super exactly once at the beginning of the method execution.
    - It must pass the parameters of the method to the super without changing them.
    - It must return the super's return value without changing it.
    - It must not refer to the super's return value. (Note 3)
    - It must not refer to the inherited state. (Note 2)
    - It must not update the inherited state.
    - It may refer to and update the state introduced by itself.

Figure 6: The MixJuice After Rule.

4.5 Other extension rules for MixJuice modules

Other extension rules for mixins described in this paper, the Plus Rule, the Functional Protocol Rule, and the Disjoint Branch Rule, can be naively modified to adapt to the MixJuice modules. All rules for mixins restrict "method extensions by mixins", which are modified to "method extensions by subclasses or sub-modules" for MixJuice.

We have already verified the three formally written modified extension rules for MixJuice.

5. THE VERIFICATION SUPPORT TOOL

We have verified eight extension rules described in this paper (four for mixins, four for MixJuice) by using a simple verification support tool written in Common Lisp. The tool generates logical formulas that should be tautological if the specifications of methods and a composability criterion are given.

Figure 7 is an example of input and output of the Plus Rule for mixins. The generated five logical formulas have been confirmed to be tautological by human examination.

Each pre- or post-condition of a method is represented as a lambda-expression, whose parameters are linearized specifications of supers and parameter names, and whose return value is a logical formula represented by an S-expression.

6. RELATED WORK

No previous research applies the notion of behavioral subtyping to the composability of mixins and aspects.

In [1], *harmless aspect composition* is defined from the viewpoint of data-flow dependency between an aspect and the original code. This definition is conservative. For example, the composition of mixins obeying the Plus Rule is judged as interference because the return value depends on all the mixins.

In [15], another *noninterference criterion* is defined. This criterion detects interference when class hierarchies are composed using a system like Hyper/J[13]. According to the definition of [15], interference means there is present a method invocation code where the method to be dispatched may be changed by the composition. This definition is conservative, because method overriding (as considered by us) may be regarded as interference even if it is the programmer's intention. On the other hand, [15] allows the change of behavior of a method if the method is known to be never invoked. From this point of view, our criterion is conservative, since we require compatibility of behavior for all methods even if they will never be invoked.

In [2], a small language modification for AspectJ is proposed to support modular reasoning. This proposition requires *accept declarations* at the place where a type of aspect called an *assistant* will be added. Therefore, existing source code should be invasively modified to add these types of aspects.

In [17], a method is described for verifying programs writ-

Input:

```
(setq plus-c1
  (list #'(lambda (R0s E0s s1 s1d s2 s2d s3 s3d p r)
            `(R1 ,s1 ,p))
        #'(lambda (R0s E0s s1 s1d s2 s2d s3 s3d p r)
            `(and (E1 ,s1 ,s1d ,p r1) (>= ,r r1)))))
(setq plus-c2
  (list #'(lambda (R0s E0s s1 s1d s2 s2d s3 s3d p r)
            `,(funcall (car R0s) (cdr R0s) (cdr E0s) s1 s1d s2 s2d s3 s3d p r))
        #'(lambda (R0s E0s s1 s1d s2 s2d s3 s3d p r)
            `(and
              ,(funcall (car E0s)(cdr R0s)(cdr E0s) s1 s1d s2 s2d s3 s3d p
                        'r02)
              (E2d ,s1d ,s2 ,s2d ,p r2)
              (>= r2 0)
              (>= ,r (+ r02 r2))))))
(setq plus-c3
  (list #'(lambda (R0s E0s s1 s1d s2 s2d s3 s3d p r)
            `,(funcall (car R0s) (cdr R0s) (cdr E0s) s1 s1d s2 s2d s3 s3d p r))
        #'(lambda (R0s E0s s1 s1d s2 s2d s3 s3d p r)
            `(and
              ,(funcall (car E0s)(cdr R0s)(cdr E0s) s1 s1d s2 s2d s3 s3d p
                        'r03)
              (E3d ,s1d ,s3 ,s3d ,p r3)
              (>= r3 0)
              (>= ,r (+ r03 r3))))))
(setq mixin-criteria
      `(((1 2)(1))
        ((1 3)(1))
        ((1 2 3)(1))
        ((1 2 3)(1 2))
        ((1 2 3)(1 3))
        ))
(dolist (c mixin-criteria)
  (print (expand-criterion (list plus-c1 plus-c2 plus-c3)
                  c
                  '(s1 s1d s2 s2d s3 s3d p r))))
```

Output:

```
(AND (-> P1 P1)
     (-> P1
         (-> (AND P2 (>= R02 R1) P3 (>= R2 0) (>= R (+ R02 R2)))
             (AND P2 (>= R R1)))))
(AND (-> P1 P1)
     (-> P1
         (-> (AND P2 (>= R03 R1) P3 (>= R3 0) (>= R (+ R03 R3)))
             (AND P2 (>= R R1)))))
(AND (-> P1 P1)
     (-> P1
         (-> (AND P2 (>= R02 R1) P3 (>= R2 0) (>= R03 (+ R02 R2)) P4
                  (>= R3 0) (>= R (+ R03 R3)))
             (AND P2 (>= R R1)))))
(AND (-> P1 P1)
     (-> P1
         (-> (AND P2 (>= R02 R1) P3 (>= R2 0) (>= R03 (+ R02 R2)) P4
                  (>= R3 0) (>= R (+ R03 R3)))
             (AND P2 (>= R02 R1) P3 (>= R2 0) (>= R (+ R02 R2))))))
(AND (-> P1 P1)
     (-> P1
         (-> (AND P2 (>= R02 R1) P3 (>= R2 0) (>= R03 (+ R02 R2)) P4
                  (>= R3 0) (>= R (+ R03 R3)))
             (AND P2 (>= R03 R1) P4 (>= R3 0) (>= R (+ R03 R3))))))
```

Figure 7: An example of input and output of the verification support tool.

ten in AspectJ by applying model-checking tools to woven programs.

7. CONCLUSION AND FUTURE WORK

We have defined the composability criterion by using the notion of pre- and post-conditions and behavioral subtyping. Briefly, provided the behavior of each class after the composition is a behavioral subtype of the behavior of the class before the composition, the composition is safe.

By applying this definition, we have proved the validity of four extension rules in the case of diamond inheritance of mixins and MixJuice modules. Extension rules guarantee the composability of two mixins or modules even if they override the same method.

We have shown that mixins and MixJuice modules can simultaneously achieve a high degree of safety and a high degree of extensibility.

The following future direction of research will realize safe and easy aspect-oriented programming.

- Making a catalog of practical extension rules. It is not easy for programmers to think up and verify their own extension rules; however, selecting rules from the provided catalog will be easy. The rule catalog will contain generalized versions of the four rules described in this paper.
- Introducing new language constructs for declaring and enforcing extension rules in aspect-oriented languages. In the case of four rules described in this paper, the compiler can enforce most parts of the restrictions on method implementations. For example, restrictions about reference and updating of states can be checked by the compiler.
- Designing assertion-checking mechanisms that detect interference of aspects during runtime. Introducing assertion-checking mechanisms as in Eiffel[11] to aspect-oriented languages will enable runtime detection of violation of declared extension rules.
- Consideration of other linearization algorithms. For example, introduction of local precedence order or final methods will increase freedom of method extensions; however, they may cause link-time errors.
- Verifying extension rules in general cases that are not restricted to the diamond inheritance case studied in this paper. This proof may be implemented by induction of the number of classes and modules.

8. REFERENCES

[1] U. Aßmann. AOP with design patterns as meta-programming operators. Technical Report 28, Universität Karlsruhe, Oct. 1997.

[2] C. Clifton and G. T. Leavens. Observers and Assistants: A proposal for modular aspect-oriented reasoning. In G. T. Leavens and R. Cytron, editors, *FOAL 2002 Proceedings: Foundations of Aspect-Oriented Langauges Workshop at AOSD 2002*, number 02-06 in Technical Report, pages 33–44. Department of Computer Science, Iowa State University, Apr. 2002.

[3] K. K. Dhara and G. T. Leavens. Forcing behavioral subtyping through specification inheritance. In *Proceedings of the 18th international conference on Software engineering*, pages 258–267. IEEE Computer Society Press, 1996.

[4] T. Elrad, M. Aksits, G. Kiczales, K. Lieberherr, and H. Ossher. Discussing aspects of AOP. *Communications of the ACM*, 44(10):33–38, 2001.

[5] Y. Ichisugi. MixJuice home page. `http://staff.aist.go.jp/y-ichisugi/mj/`.

[6] Y. Ichisugi and A. Tanaka. Difference-based modules: A class-independent module mechanism. In *Proc. of the ECOOP2002*, LNCS 2374, pages 62–88, 2002.

[7] G. Kiczales, J. des Rivieres, and D. G. Bobrow. *The Art of Metaobject Protocol*. MIT Press, 1991.

[8] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J. Loingtier, and J. Irwin. Aspect-oriented programming. In *Proc. of the ECOOP'97*, LNCS 1241, pages 220–242, 1997. Invited Talk.

[9] B. H. Liskov and J. M. Wing. A behavioral notion of subtyping. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 16(6):1811–1841, 1994.

[10] R. C. Martin. The Liskov Substitution Principle. *C++ Report*, Mar 1996.

[11] B. Meyer. *Object-Oriented Software Construction, 2nd Ed.* Prentice-Hall, Inc., 1997.

[12] D. A. Moon. Object-oriented programming with Flavors. In *Conference proceedings on Object-oriented programming systems, languages and applications*, pages 1–8. ACM Press, 1986.

[13] H. Ossher and P. Tarr. Multi-dimensional separation of concerns and the hyperspace approach. In *Proc. of the Symposium on Software Architectures and Component Technology: The State of the Art in Software Development, Kluwer*, 2000.

[14] C. Ruby and G. T. Leavens. Safely creating correct subclasses without seeing superclass code. In *Proceedings of the conference on Object-oriented programming, systems, languages, and applications*, pages 208–228. ACM Press, 2000.

[15] G. Snelting and F. Tip. Semantic-based composition of class hierarchies. In *Proc. of the ECOOP2002*, LNCS 2374, pages 562–584, 2002.

[16] G. Steele. *Common Lisp the Language, 2nd edition*. Digital Press, 1990.

[17] N. Ubayashi and T. Tamai. Aspect oriented programming with model checking. In *Proc. of AOSD 2002(1st International Conference on Aspect-Oriented Software Development)*, pages 148–154, Apr. 2002.