Semantics-Based Composition of Class Hierarchies

Gregor Snelting[1] and Frank Tip[2]

[1] Universität Passau, Lehrstuhl für Softwaresysteme
Innstr. 33, 94032 Passau, Germany
snelting@fmi.uni-passau.de
[2] IBM T.J. Watson Research Center
P.O. Box 704, Yorktown Heights, NY 10598, USA
tip@watson.ibm.com

Abstract. Class hierarchy composition aims at constructing software systems by *composing* a set of class hierarchies into an executable system. Current composition algorithms cannot provide semantic guarantees. We present a composition algorithm, together with an interference criterion and its correctness proof, which guarantees that behavior of the original hierarchies is preserved for interference-free compositions. In case of interference, an impact analysis can determine the consequences of integration. The method is based on existing program analysis technology and is illustrated by various examples.

1 Introduction

Class hierarchy composition aims at constructing software systems by *composing* the code associated with a set of class hierarchies into an executable system [5, 19], or by weaving separately written aspects into a class hierarchy [8, 22]. Advocates of composition argue that, by putting the functionality associated with each system feature in a separate hierarchy, a better separation of concerns is achieved, resulting in code that is easier to understand, maintain, and change.

Although considerable amounts of work have been devoted to developing specification formalisms for software composition, and on the methodological aspects of compositional software development, current techniques and tools for hierarchy composition operate on a purely syntactical basis and cannot provide any semantic guarantees about the behavior of the composed hierarchy. It is thus our aim to develop a semantically well-founded notion of composition that enables reasoning about the behavior of composed class hierarchies. We have opted for the following approach:

– We define notions of *static interference* and *dynamic interference* that capture how features in one hierarchy may impact the behavior of code in another. The former notion captures behavioral impact at composition-time, whereas the latter is concerned with run-time changes in program behavior.

$\mathcal{H}_1$

```
class Course {
  Course(Professor p, String name){
    prof = p; courseName = name;
    students = new HashSet();
  }
  String toString(){ return courseName; }
  void enroll(Student s){
    if (!students.contains(s)){
      students.add(s);
      s.coursesTaken.add(this); }
  }
  void assign(Professor p){
    prof = p;
    p.coursesGiven.add(this);}
  Set students; Professor prof;
  String courseName;
}
class Person {
  Person(String n, String a){
    name = n; address = a;
  }
  String name; String address; }
class Student extends Person {
  Student(String n, String a){
    super(n, a);
    coursesTaken = new HashSet(); }
  String toString(){
    return "student "+name+" takes "+
        coursesTaken+"\n"; }
  Set coursesTaken;
}
class Professor extends Person {
  Professor(String n, String a){
    super(n, a);
    coursesGiven = new HashSet(); }
  String toString(){
    return "prof. "+name+" teaches "+
        coursesGiven+"\n"; }
  boolean approveGraduation(Student s){
    return approveCourses(s); }
  boolean approveCourses(Student s){
    return true; // details omitted
  }
  Set coursesGiven;
}
class Driver1 {
  void main(){
    Professor p1 =
      new Professor("prof1","padd1");  [P1]
    Student s1 =
      new Student("stu1","sadd1");  [S1]
    Course c1 = new Course(p1, "CS121");
    c1.enroll(s1); c1.assign(p1);
    boolean b = p1.approveGraduation(s1);
  }
}
```

$\mathcal{H}_2$

```
class Person { ··· } // as in H1
class Student {
  Student(String n, String a){
    ··· // as in H1
  }
  void setAdvisor(Professor p){
    advisor = p; }
  Set coursesTaken; Professor advisor;
}
class Professor {
  Professor(String n, String a){
    ··· // as in H1
  }
  void hireAssistant(Student s){
    assistant = s; }
  Set coursesGiven; Student assistant;
}
class Driver2 {
  void main(){
    Professor p2 =
      new Professor("prof2","padd2");  [P2]
    Student s2 =
      new Student("stu2","sadd2");  [S2]
    s2.setAdvisor(p2);p2.hireAssistant(s2);
  }
}
```

$\mathcal{H}_3$

```
class Person { ··· } // as in H1
class Student {
  Student(String n,String a){
    ··· // as in H1
  }
  Set coursesTaken;
}
class PhDStudent extends Student {
  PhDStudent(String n,String a){
    super(n,a);
  }
}
class Professor {
  Professor(String n,String a){
    ··· // as in H1
  }
  boolean approveGraduation(Student s){
    boolean approved = approveCourses(s);
    if (s instanceof PhDStudent){
      approved = approved &&
        approveThesis((PhDStudent)s);
    }
    return approved;
  }
  boolean approveCourses(Student s){
    ··· // as in H1
  }
  boolean approveThesis(PhDStudent s){
    /* details omitted */
  }
  Set coursesGiven;
}
class Driver3 {
  void main(){
    Professor p3 =
      new Professor("prof3","padd3");  [P3]
    PhDStudent s3 =
      new PhDStudent("stu3","sadd3");  [S3]
    p3.approveGraduation(s3);
  }
}
```

Fig. 1. Example hierarchies concerned with different aspects of university life. Allocation sites are labeled (shown in boxes).

```
class Course { ··· }                                  // as in H1
class Person { ··· }                                  // as in H1 and H2
class Student extends Person {
  Student(String n, String a){ ··· }                  // as in H1 and H2
  String toString(){ ··· }                            // as in H1
  void setAdvisor(Professor p){ ··· }                 // as in H2
  Set coursesTaken;
  Professor advisor;
}
class Professor extends Person {
  Professor(String n, String a){ ··· }                // as in H1 and H2
  String toString(){ ··· }                            // as in H1
  boolean approveGraduation(Student s){ ··· }         // as in H1
  boolean approveCourses(Student s){ ··· }            // as in H1
  void hireAssistant(Student s){ ··· }                // as in H2
  Set coursesGiven;
  Student assistant;
}
class Driver1 { ··· }                                 // as in H1
class Driver2 { ··· }                                 // as in H2
```

Fig. 2. A basic composition: $\mathcal{H}_1 \oplus \mathcal{H}_2$.

- We consider two kinds of compositions. *Basic compositions* involve hierarchies that do not statically interfere. *Overriding compositions* rely on a mechanism by which a user can explicitly resolve static interference.
- In cases where interference is found, an *impact analysis* (similar to the one of [16]) is performed that determines a set of methods in the composed hierarchy for which preservation of behavior cannot be guaranteed.

Hence, our techniques allow a developer to quickly determine if a proposed composition may result in behavioral changes, and—if so—report precisely which parts of the program may be affected. In the longer term, we hope to incorporate these techniques in a system such as Hyper/J [19] in the form of a tool that performs various sanity checks on compositions.

In order to illustrate our techniques, Figure 1 shows three hierarchies that model a number of aspects of university life. $\mathcal{H}_1$ defines classes `Course`, `Person`, `Student`, and `Professor`, and provides functionality for enrolling students in courses, for associating professors with courses, and for professors to approve the graduation of students (method `Professor.approveGraduation()`). This latter operation requires the approval of the courses taken by a student, modeled using a method `approveCourses()`. We have omitted the details of `approveCourses()`, but one can easily imagine adding functionality for keeping track of a student's course load and grades which would be checked by the professor to base his decision on. Class `Driver1` contains a small test driver that exercises the functionality of hierarchy $\mathcal{H}_1$.

Hierarchy $\mathcal{H}_2$ is concerned with employment and advisory relationships. A student can designate a professor as his/her advisor (method

Student.setAdvisor(), and a professor can hire a student as a teaching assistant using method Professor.hireAssistant(). Class Driver2 exercises the functionality in this hierarchy.

Hierarchy $\mathcal{H}_3$ shows a slightly more elaborate model, where a distinction is made between (undergraduate) Students and PhDStudents. This impacts the approval of graduations, because PhDStudents are also required to produce a thesis of sufficient quality (modeled by method Professor.approvePhDThesis()). Due to space limitations, we have omitted the details of this method.

```
class Course { ··· }                                  // as in H1
class Person { ··· }                                  // as in H1 and H3
class Student extends Person {
  Student(String n, String a){ ··· }                  // as in H1 and H3
  String toString(){ ··· }                            // as in H1
  Set coursesTaken;
}
class PhDStudent extends Student { ··· }              // as in H3
class Professor extends Person {
  Professor(String n, String a){ ··· }                // as in H1 and H3
  String toString(){ ··· }                            // as in H1
  boolean approveGraduation(Student s){ ··· }         // as in H3
  boolean approveCourses(Student s){ ··· }            // as in H1 and H3
  boolean approveThesis(PhDStudent s){ ··· }          // as in H3
  Set coursesGiven;
}
class Driver1 { ··· }                                 // as in H1
class Driver3 { ··· }                                 // as in H3
```

Fig. 3. An overriding composition: $\mathcal{H}_1 \oplus \mathcal{H}_3$.

Let us now consider the composition of $\mathcal{H}_1$ and $\mathcal{H}_2$. These hierarchies are not disjoint, since they contain the same classes. However, since there are no "syntactic collisions" between members in $\mathcal{H}_1$ and $\mathcal{H}_2$ (i.e., $\mathcal{H}_1$ and $\mathcal{H}_2$ do not contain methods with the same name and signatures, but with different bodies), one can simply construct a hierarchy that contains the union of the classes in $\mathcal{H}_1$ and $\mathcal{H}_2$, where each class in the combined hierarchy contains the union of the methods/fields that occur in the corresponding class(es) in $\mathcal{H}_1$ and $\mathcal{H}_2$. The resulting hierarchy is shown in Figure 2. What can be said about the behavior of $\mathcal{H}_1 \oplus \mathcal{H}_2$? In this case, our interference check guarantees that the behavior of the client applications of these hierarchies (modeled by Driver1.main() and Driver2.main()) are unaffected by the composition. For this specific composition, we can even provide the stronger guarantee that the behavior of *any* client of $\mathcal{H}_1$ and $\mathcal{H}_2$ is preserved. As we shall see shortly, this is not always the case.

Now consider composing $\mathcal{H}_1$ and $\mathcal{H}_3$, which contain different methods Professor.approveGraduation(), an example of static interference. Method approveGraduation() in $\mathcal{H}_3$ is "more general" than approveGraduation() in $\mathcal{H}_1$. In constructing $\mathcal{H}_1 \oplus \mathcal{H}_3$ (see Figure 3), we have assumed that the user spec-

ified that the definition of `approveGraduation()` in $\mathcal{H}_3$ should be preferred over that in $\mathcal{H}_1$. In this case, our techniques report dynamic interference, i.e., preservation of behavior for clients of the original hierarchies cannot be guaranteed. Impact analysis reports that `Driver3.main()` is not affected by the composition, but that the behavior of `Driver1.main()` may have changed.

2 Composition of hierarchies

One of the first issues that arises when composing two class hierarchies is the question which classes and methods in the input hierarchies correspond. The Hyper/J composition system [19] relies on a specification language to express these correspondences. For example, one can specify "merge-by-name" compositions in which two classes in different input hierarchies are matched if they have the same name, and one can explicitly specify pairs of matching classes (with different names) using an "equate" construct.

In order to simplify the presentation in this paper, we will assume that classes are matched "by name" only. Compositions that are not name-based can be modeled using an additional preprocessing step in which classes and methods are renamed appropriately. In particular, manually established relations between entities in the two hierarchies are assumed to be modeled by appropriate renaming.

2.1 Class hierarchies

Definition 1 defines the notion of a class hierarchy. To keep our definitions simple, we assume that fields and abstract methods have undefined bodies ($body(m) = \bot$), and that fields and abstract methods cannot have the same name.

Definition 1 (class hierarchy). *A class hierarchy $\mathcal{H}$ is a set of classes together with an inheritance relation: $\mathcal{H} = (\mathcal{C}, \leq)$. For a class $C \in \mathcal{C}$ we also write $C \in \mathcal{H}$. A class $C \in \mathcal{H}$ has a name and contains a set of members[1]: $C = (n, M)$, where $\text{name}(C) = n, \text{members}(C) = M$. A member $m \in \text{members}(C)$ is characterized by its name, its signature and its body: $m = (f, \sigma, B)$ where $\sigma \in \mathcal{C}^* \times \mathcal{C}$. We will use $\text{namesig}(m)$ to denote the combination $\langle f, \sigma \rangle$ that together uniquely identify a member within a class, and $\text{body}(m)$ to denote the body B of member m.*

2.2 Classes and inheritance relations in the composed hierarchy

Semantically sound composition requires that the original inheritance relations can be order-embedded into the composed hierarchy. That is, a relationship `A instanceof B` that holds in an input hierarchy should also hold in the composed hierarchy. In general, one cannot simply compute the union of the inheritance relations in the input hierarchies because the resulting hierarchy may contain

[1] According to this definition, $members(C)$ does not contain inherited members that are declared in superclasses of C.

cycles. We therefore use a well-known factorization technique (see, e.g., [3]) that produces an acyclic hierarchy. This construction has the advantage that hierarchies can be composed even if there are cycles in the union of the original inheritance relations – which might sometimes be useful in practice.

Given two input hierarchies $\mathcal{H}_1$ and $\mathcal{H}_2$, their composition is denoted $\mathcal{H}_1 \oplus \mathcal{H}_2$. The construction of $\mathcal{H}_1 \oplus \mathcal{H}_2$ is given in Definition 2. This involves: creating a set of pairs of the form $\langle$ class, hierarchy $\rangle$ (step 1), determining the "union" of the inheritance relations in the input hierarchies (assuming that classes are matched by name) (2), determining cycles in the transitive closure of these inheritance relations, and constructing a set of equivalence classes[2] $\mathcal{E}$ corresponding to these cycles (3-5), creation of a class in the composed hierarchy for each equivalence class in $\mathcal{E}$(6), associating a name and a set of members (7) with each class, and creation of the inheritance relations in the composed hierarchy (8).

Definition 2 (hierarchy composition). *Let $\mathcal{H}_1 = (\mathcal{C}_1, \leq_1)$ and $\mathcal{H}_2 = (\mathcal{C}_2, \leq_2)$ be two class hierarchies. Then, $\mathcal{H}_1 \oplus \mathcal{H}_2 = (\mathcal{C}, \leq)$, which is defined as follows:*

1. $S = \{ \langle C_1, \mathcal{H}_1 \rangle \mid C_1 \in \mathcal{C}_1 \} \cup \{ \langle C_2, \mathcal{H}_2 \rangle \mid C_2 \in \mathcal{C}_2 \}$,
2. $\langle C_1, \mathcal{H}_1 \rangle \leq' \langle C_2, \mathcal{H}_1 \rangle \Leftarrow C_1 \leq_1 C_2$, $\langle C_1, \mathcal{H}_2 \rangle \leq' \langle C_2, \mathcal{H}_2 \rangle \Leftarrow C_1 \leq_2 C_2$,
   $\langle C_1, \mathcal{H}_i \rangle \leq' \langle C_2, \mathcal{H}_j \rangle \Leftarrow \mathrm{name}(C_1) = \mathrm{name}(C_2)$, $(i, j \in \{ 1,\ 2 \})$
3. $x \, \rho \, y \iff x \leq'^* y \wedge y \leq'^* x$,
4. $\leq \;=\; \leq'^* / \rho$,
5. $\mathcal{E} = \{ [x]_\rho \mid x \in S \}$,
6. $\mathcal{C} = \{ \mathrm{class}([x]_\rho) \mid [x]_\rho \in \mathcal{E} \}$,
7. $\mathrm{class}([x]_\rho) = \langle \mathrm{name}([x]_\rho), \mathrm{members}([x]_\rho) \rangle$, *and*
8. $\mathrm{class}([x]_\rho) \leq \mathrm{class}([y]_\rho) \iff [x]_\rho \leq [y]_\rho$

The name *function determines the name of the composed class from the names of the classes in equivalence class $[C]_\rho$ and will not be formally defined here (some examples will be given below). Note that the members of a composed class do not include inherited members from the original classes, but only the members defined locally in the original classes. Different* members *operators will be presented for different kinds of compositions in Definitions 4 and 6 below.*

Note that $\leq'$ is not necessarily transitive, hence the use of the closure operator. As usual, $[x]_\rho$ consists of all classes ρ-equivalent to x, and $[x]_\rho \leq [y]_\rho \iff x \leq' y$; we assume that $\leq$ is the smallest partial order satisfying the conditions from the definition. We will use $\mathcal{E}$ to denote the set of all equivalence classes $[\langle C, \mathcal{H} \rangle]_\rho$, for any $\langle C, \mathcal{H} \rangle \in S$. Moreover, we define a partial order on $\mathcal{E}$ by: $\langle C, \mathcal{H} \rangle \leq' \langle C', \mathcal{H}' \rangle \iff [\langle C, \mathcal{H} \rangle]_\rho \leq [\langle C', \mathcal{H}' \rangle]_\rho$. Intuitively, one can imagine this order as the directed acyclic graph of all strongly connected components of the union of the two input hierarchies.

[2] Unfortunately, the term "class" as in "equivalence class" is different from "class" as in "class hierarchy"; we use this "overloading" in order to be compatible with both mathematical and computer science conventions.

Example 1. In the hierarchy of Figure 4(a), the transitive closure of $\leq_1$ and $\leq_2$ does not contain any cycles. Hence, we have: $\mathcal{E} = \{ \{\langle A, \mathcal{H}_1\rangle, \langle A, \mathcal{H}_2, \rangle\}, \{\langle B, \mathcal{H}_1\rangle\}, \{\langle C, \mathcal{H}_1\rangle, \langle C, \mathcal{H}_2, \rangle\}, \{\langle D, \mathcal{H}_1\rangle\}, \{\langle E, \mathcal{H}_1\rangle\}, \{\langle F, \mathcal{H}_2, \rangle\}, \{\langle G, \mathcal{H}_2, \rangle\} \}$. Consequently, the following inheritance relations are constructed: $B < A, F < A, C < F, G < F, D < C, E < C$. Here, class names in the composed hierarchy are generated from the names of the classes in the corresponding equivalence sets (e.g., class A corresponds to $\{ \langle A, \mathcal{H}_1\rangle, \langle A, \mathcal{H}_2\rangle \}$). Note that *immediate* subclass/superclass relations need not be preserved: F is now between A and C. □

Example 2. For the slightly more interesting example of Figure 4(b), we have: $\mathcal{E} = \{ \{\langle A, \mathcal{H}_1\rangle, \langle A, \mathcal{H}_2\rangle, \langle C, \mathcal{H}_1\rangle, \langle C, \mathcal{H}_2\rangle, \langle F, \mathcal{H}_2\rangle\}, \{\langle B, \mathcal{H}_1\rangle\}, \{\langle D, \mathcal{H}_1\rangle\}, \{\langle E, \mathcal{H}_1\rangle\}, \{ \langle G, \mathcal{H}_2, \rangle\} \}$. The composed hierarchy contains a class for each of these equivalence classes (for the purposes of this example, we assume that the *name* function constructs a class name by concatenating the names of elements in the equivalence class). There is an inheritance relation $X < Y$ if the equivalence class corresponding to X contains a class x, and the equivalence class corresponding to Y contains a class y such that x inherits from y in one or both of the input hierarchies. For example, class B inherits from ACF because $\langle B, \mathcal{H}_1\rangle$ is part of equivalence class $\{ \langle B, \mathcal{H}_1\rangle \}$, $\langle A, \mathcal{H}_1\rangle$ is part of equivalence class $\{ \langle A, \mathcal{H}_1\rangle, \langle A, \mathcal{H}_2\rangle, \langle C, \mathcal{H}_1\rangle, \langle C, \mathcal{H}_2\rangle, \langle F, \mathcal{H}_2\rangle \}$, and class B inherits from class A in $\mathcal{H}_1$. Again, *immediate* subclass/superclass relations need not be preserved: some immediate relations (e.g., $F < C$) have been collapsed. □

In case classes have been merged, new class names have been introduced as well (e.g., ACF in the above example). Thus any client code must be transformed accordingly: any occurrence of the old class name x must be replaced by $\mathit{name}([x]_\rho)$. In the example, any occurrence of class names A, C, or F in client code must be replaced by ACF.

A final issue to note is that the composed inheritance relation may contain multiple inheritance. This may cause problems in languages such as Java that do not support general multiple inheritance. We consider this issue to be outside the scope of this paper, and plan to pursue an approach in which multiple inheritance is automatically transformed into delegation, along the lines of [20].

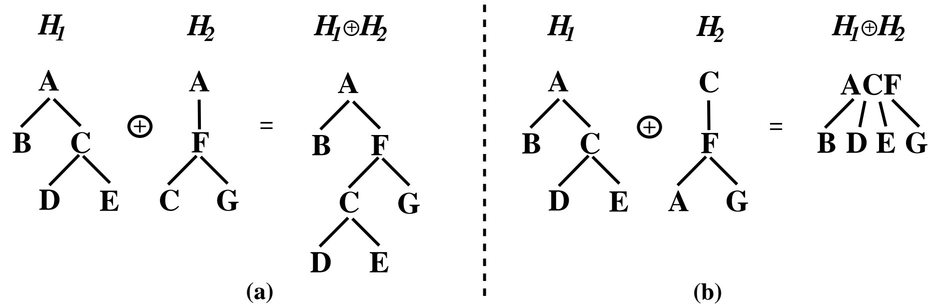

Fig. 4. Hierarchy composition without class merging(a), and with class merging (b).

2.3 Basic composition

In defining the set of members in the composed hierarchy, the question arises of what to do when two or more classes in an equivalence class $[x]_\rho$ define the same member. We will refer to such cases as *static interference*. The easiest approach of dealing with this issue is to simply disallow composition if static interference occurs. We will refer to this scenario as *basic composition*. As we shall see shortly, the absence of static interference does not guarantee preservation of behavior.

Definition 3 defines basic static interference. Note that it *does* allow situations where an equivalence class $[x]_\rho$ contains two elements $\langle C_1, \mathcal{H}_1 \rangle$ and $\langle C_2, \mathcal{H}_2 \rangle$ such that C_1 and C_2 each contain a member m, provided that (i) $\mathit{body}(m) = \bot$ holds for at least one of these m's, or (ii) that the two m's have the same body.

Definition 3 (basic static interference). *$\mathcal{E}$ contains basic static interference if there is an equivalence class $[x]_\rho \in \mathcal{E}$ such that for some $\langle C_1, \mathcal{H} \rangle, \langle C_2, \mathcal{H}' \rangle \in [x]_\rho$, $C_1 \neq C_2$, $m_1 \in \mathit{members}(C_1)$, $m_2 \in \mathit{members}(C_2)$, $\mathit{namesig}(m_1) = \mathit{namesig}(m_2)$ we have that: $\mathit{body}(m_1) \neq \bot$, $\mathit{body}(m_2) \neq \bot$, and $\mathit{body}(m_1) \neq \mathit{body}(m_2)$.*

Definition 4 defines the set of members in the composed hierarchy. Note that, in cases where the classes in an equivalence class contain multiple methods with the same name and signature, the unique method with a non-$\bot$ body is selected.

Definition 4 (members). *Let $\mathcal{E}$ be free of basic static interference, and let $[x]_\rho \in \mathcal{E}$ be an equivalence class. Define:*

$$\begin{aligned} \mathit{members}([x]_\rho) = \quad & \{\, m \mid \langle C, \mathcal{H} \rangle \in [x]_\rho,\ m \in \mathit{members}(C), \\ & \quad (\langle C', \mathcal{H}' \rangle \in [x]_\rho,\ C' \neq C,\ m' \in \mathit{members}(C'), \\ & \quad \mathit{namesig}(m) = \mathit{namesig}(m') \implies \mathit{body}(m') = \bot) \,\} \end{aligned}$$

Example 3. The transitive closure of the inheritance relations in $\mathcal{H}_1$ and $\mathcal{H}_2$ of Figure 1 does not contain any cycles. Hence, the construction of Definition 2 produces a hierarchy with classes `Course`, `Person`, `Student`, `Professor`, `Driver1`, and `Driver2`, with inheritance relations `Student` $<$ `Person` and `Professor` $<$ `Person`. The equivalence classes constructed are: $S_1 = \{ \langle \texttt{Course}, \mathcal{H}_1 \rangle \}$, $S_2 = \{ \langle \texttt{Person}, \mathcal{H}_1 \rangle \}$, $S_3 = \{ \langle \texttt{Student}, \mathcal{H}_1 \rangle, \langle \texttt{Student}, \mathcal{H}_2 \rangle \}$, $S_4 = \{ \langle \texttt{Professor}, \mathcal{H}_1 \rangle, \langle \texttt{Professor}, \mathcal{H}_2 \rangle \}$, $S_5 = \{ \langle \texttt{Driver1}, \mathcal{H}_1 \rangle \}$, and $S_6 = \{ \langle \texttt{Driver2}, \mathcal{H}_2 \rangle \}$. Definition 3 states that there is basic static interference if an equivalence class S contains multiple methods with the same name but different bodies. Singleton equivalence classes such as S_1, S_2, S_5, and S_6 cannot give rise to interference because a class can contain only one method with a given name and signature. S_3 and S_4 do not give rise to interference either because `Student` and `Professor` in $\mathcal{H}_1$ and $\mathcal{H}_2$ do not contain conflicting methods. Hence, there is no basic static interference. Figure 2 shows the composed hierarchy. □

Example 4. Consider composing the class hierarchies $\mathcal{H}_1$ and $\mathcal{H}_3$ of Figure 1. The set equivalence classes $\mathcal{E}$ constructed according to Definition 2 contains,

among others, the element $S = \{ \langle \texttt{Professor}, \mathcal{H}_1 \rangle, \langle \texttt{Professor}, \mathcal{H}_3 \rangle \}$. $\mathcal{E}$ exhibits basic static interference because both elements of S contain a member `Professor.approveGraduation(Student)` and the bodies of these methods are different. Hence, basic composition cannot be applied to $\mathcal{H}_1$ and $\mathcal{H}_3$. □

2.4 Overriding composition

As we have seen in Figure 3, basic static interference is not necessarily an unwanted phenomenon. Often, a method from $\mathcal{H}_2$ is an "improved" or "generalized" version of a method in $\mathcal{H}_1$. To address such cases, we augment basic composition with a mechanism that allows one to express conditions such as "member $B.m$ in $\mathcal{H}'$ has precedence over member $A.m$ in $\mathcal{H}$". This is captured by a precedence relation $\ll$ containing elements $\langle \mathcal{H}, m_1 \rangle \ll \langle \mathcal{H}', m_2 \rangle$ indicating that method m_2 of hierarchy $\mathcal{H}'$ has precedence over method m_1 of hierarchy $\mathcal{H}$. Note that it may be the case that $\mathcal{H} = \mathcal{H}'$. It is assumed that '$\ll$' is a partial order.

The static interference notion of Definition 3 only requires minor modifications to allow situations where an equivalence class contains two classes C_1 and C_2 originating from hierarchies $\mathcal{H}_1$ and $\mathcal{H}_2$, respectively, such that $\langle \mathcal{H}_1, m_1 \rangle$ and $\langle \mathcal{H}_2, m_2 \rangle$ are $\ll$-ordered. Definition 5 shows the resulting definition.

Definition 5 (overriding static interference). *$\mathcal{E}$ contains overriding static interference w.r.t. '$\ll$' if there is an equivalence class $[x]_\rho \in \mathcal{E}$ such that for some $\langle C_1, \mathcal{H} \rangle$, $\langle C_2, \mathcal{H}' \rangle \in [x]_\rho$, $C_1 \neq C_2$, $m_1 \in members(C_1)$, $m_2 \in members(C_2)$, and $namesig(m_1) = namesig(m_2)$ we have that: $body(m_1) \neq \bot$, $body(m_2) \neq \bot$, $body(m_1) \neq body(m_2)$, $\langle \mathcal{H}, m_1 \rangle \not\ll \langle \mathcal{H}', m_2 \rangle$, and $\langle \mathcal{H}', m_2 \rangle \not\ll \langle \mathcal{H}, m_1 \rangle$.*

Definition 6 shows the *members* function for overriding compositions. In the sequel, we will often say "$\mathcal{H}_1 \oplus \mathcal{H}_2$ is free of overriding syntactic interference" if it is obvious which ordering '$\ll$' is used.

Definition 6 (members). *Let $\mathcal{E}$ be free of overriding static interference w.r.t. '$\ll$', and let $[x]_\rho \in \mathcal{E}$ be an equivalence class. Define:*

$$\begin{aligned} members([x]_\rho) = \; & \{ m_1 \mid \langle C_1, \mathcal{H}_1 \rangle \in [x]_\rho,\ m_1 \in members(C_1), \\ & \quad (\langle C_2, \mathcal{H}_2 \rangle \in [x]_\rho,\ C_2 \neq C_1,\ m_2 \in members(C_2), \\ & \quad namesig(m_1) = namesig(m_2)) \implies \\ & \quad\quad body(m_2) = \bot \vee \langle \mathcal{H}_2, m_2 \rangle \ll \langle \mathcal{H}_1, m_1 \rangle \; \} \end{aligned}$$

Example 5. Consider an overriding composition of hierarchies $\mathcal{H}_1$ and $\mathcal{H}_3$ of Figure 1 using $\langle$`Professor.approveGraduation(Student)`, $\mathcal{H}_1 \rangle \ll \langle$`Professor.approveGraduation(Student)`, $\mathcal{H}_3 \rangle$. Then, the set of equivalence classes $\mathcal{E}$ constructed by Definition 2 is: $S_1 = \{ \langle \texttt{Course}, \mathcal{H}_1 \rangle \}$, $S_2 = \{ \langle \texttt{Person}, \mathcal{H}_1 \rangle \}$, $S_3 = \{ \langle \texttt{Student}, \mathcal{H}_1 \rangle \}$, $S_4 = \{ \langle \texttt{Professor}, \mathcal{H}_1 \rangle, \langle \texttt{Professor}, \mathcal{H}_3 \rangle \}$, $S_5 = \{ \langle \texttt{Driver1}, \mathcal{H}_1 \rangle \}$, $S_6 = \{ \langle \texttt{PhDStudent}, \mathcal{H}_3 \rangle \}$, and $S_7 = \{ \langle \texttt{Driver3}, \mathcal{H}_3 \rangle \}$. Since singleton sets never give rise to interference, we only need to verify that S_4 does not cause overriding static interference. This is the case because the only method that occurs in both `Professor` classes is `approveGraduation()`, and these methods are $\ll$-ordered. The composed hierarchy can now be constructed using Definition 6, and was shown earlier in Figure 3. □

2.5 Type correctness

A class hierarchy is type correct if: (1) any member access $e.m(\cdots)$ refers to a declared member definition, and (2) for any assignment $x = y$, the type of x is a superclass of the type of y. As a first step towards providing semantic guarantees about the composed class hierarchy, we demonstrate that the composed hierarchy is type correct. Due to space limitations, we only demonstrate these properties for basic compositions. The arguments for overriding compositions are similar.

Definition 7 (type correctness). *Let $\mathcal{H}$ be a hierarchy.*

1. *The static type of an object or object reference o in a hierarchy is denoted $TypeOf(\mathcal{H}, o)$. For convenience, we use $TypeOf(\mathcal{H}_{1,2}, o) = C$ as an abbreviation for $TypeOf(\mathcal{H}_1, o) = C \lor TypeOf(\mathcal{H}_2, o) = C$.*
2. *For a class $C \in \mathcal{H}$ and $m \in members(C)$, we define $StaticLookup(\mathcal{H}, C, m) = m'$, where $m' \in members(C')$ for some class C' such that $C \leq C'$, $\mathrm{namesig}(m) = \mathrm{namesig}(m')$, and there is no class C'' such that $C \leq C'' \leq C'$, $m'' \in members(C'')$, $\mathrm{namesig}(m) = \mathrm{namesig}(m'')$. We will use $StaticLookup(\mathcal{H}_{1,2}, C, m) = m'$ as a shorthand for $StaticLookup(\mathcal{H}_1, C, m) = m' \lor StaticLookup(\mathcal{H}_2, C, m) = m'$.*
3. *A hierarchy $\mathcal{H}$ is type correct if for all assignments $x = y \in \mathcal{H}$ we have that $TypeOf(\mathcal{H}, x) \geq TypeOf(\mathcal{H}, y)$, and for all member accesses $o.m(\cdots) \in \mathcal{H}$ we have that: $StaticLookup(\mathcal{H}, TypeOf(\mathcal{H}, o), m) \neq \bot$.*

Note that if $TypeOf(\mathcal{H}_{1,2}, o) = C$, then by construction $TypeOf(\mathcal{H}_1 \oplus \mathcal{H}_2, o) = class([C]_\rho)$. As an example, consider Figure 4, and assume that v is a variable such that $TypeOf(\mathcal{H}_1, v) = A$. Then, in Figure 4(a) we have that $TypeOf(\mathcal{H}_1 \oplus \mathcal{H}_2, v) = A$ and in Figure 4(b) that $TypeOf(\mathcal{H}_1 \oplus \mathcal{H}_2, v) = ACF$. The latter case demonstrates that sometimes new class names are introduced, and member declarations must be changed accordingly. In particular, whenever $name([C]_\rho) \neq name(C)$, all declarations containing class name C must be updated to reflect the new class name $name([C]_\rho)$. This will only happen if classes have been merged due to cycles in the transitive inheritance relations of the input hierarchies.

The following two lemmas show that assignments and member lookups in the composed hierarchy remain type correct. Note that this includes assignments due to parameter-passing in method calls, and implicit assignments to `this`-pointers.

Lemma 1 (assignment correctness). *Let $x = y$ be an assignment in $\mathcal{H}_{1,2}$. Then, this assignment is still type correct in $\mathcal{H}_1 \oplus \mathcal{H}_2$.*

Proof. Without loss of generality, let $x = y \in \mathcal{H}_1$. Then, $TypeOf(\mathcal{H}_1, x) \geq_1 TypeOf(\mathcal{H}_1, y)$. By construction, $TypeOf(\mathcal{H}_1 \oplus \mathcal{H}_2, x) = class([TypeOf(\mathcal{H}_1, x)]_\rho) \geq class([TypeOf(\mathcal{H}_1, y)]_\rho) = TypeOf(\mathcal{H}_1 \oplus \mathcal{H}_2, y)$. □

Lemma 2 (member access correctness). *If $\mathcal{H}_1$ and $\mathcal{H}_2$ are type correct and without basic static interference, then $StaticLookup(\mathcal{H}_{1,2}, TypeOf(\mathcal{H}_{1,2}, o), m) \neq \bot \implies StaticLookup(\mathcal{H}_1 \oplus \mathcal{H}_2, TypeOf(\mathcal{H}_1 \oplus \mathcal{H}_2, o), m) \neq \bot$*

Proof. Without loss of generality, let $C = \mathit{TypeOf}(\mathcal{H}_1, o)$, and let D be the class which contains $m' = \mathit{StaticLookup}(\mathcal{H}_1, \mathit{TypeOf}(\mathcal{H}_1, o), m)$. Then, we have $C \leq_1 D$, $m \in \mathit{members}(D)$ and by construction $m \in \mathit{members}(\mathit{class}([D]_\rho))$ as there is no static interference. Furthermore, $\mathit{class}([C]_\rho) \leq \mathit{class}([D]_\rho)$ and $\mathit{TypeOf}(\mathcal{H}_1 \oplus \mathcal{H}_2, o) = \mathit{class}([C]_\rho)$. m could also occur in a subclass of $\mathit{class}([D]_\rho)$ in $\mathcal{H}_1 \oplus \mathcal{H}_2$, but for class D' which contains $m'' = \mathit{StaticLookup}(\mathcal{H}_1 \oplus \mathcal{H}_2, \mathit{TypeOf}(\mathcal{H}_1 \oplus \mathcal{H}_2, o), m)$, we have in any case that $D' \leq \mathit{class}([D]_\rho)$, hence $\mathit{StaticLookup}(\mathcal{H}_1 \oplus \mathcal{H}_2, \mathit{TypeOf}(\mathcal{H}_1 \oplus \mathcal{H}_2, o), m) \neq \bot$. $\square$

Corollary 1. *For hierarchies $\mathcal{H}_1$ and $\mathcal{H}_2$ without static interference, $\mathcal{H}_1 \oplus \mathcal{H}_2$ is type correct.*

```
class A {
  void foo(){ ··· }
}
class B extends A {
  /* no foo() */
}
class C {
  static void main(){
    A o = new B(); [B1]
    o.foo();
  }
}
```
$\mathcal{H}_1$

```
class A {
  void foo(){ /* same as H1 */ }
}
class B extends A {
  void foo(){ ··· }
}
```
$\mathcal{H}_2$

```
class A {
  void foo(){ ··· }
}
class B extends A {
  void foo(){ /* from H2 */ }
}
class C {
  static void main(){
    A o = new B(); [B1]
    o.foo();
  }
}
```
$\mathcal{H}_1 \oplus \mathcal{H}_2$

Fig. 5. Dynamic interference in a basic composition.

3 Dynamic interference

3.1 Motivating examples

Even basic composition (which makes the strongest assumptions about static noninterference) does not guarantee preservation of client behavior. This can be seen in the basic composition of Figure 5, which does not exhibit static interference. However, $\mathcal{H}_1 \oplus \mathcal{H}_2$ contains an overriding definition of `foo()` in class `B`, hence the call to `foo()` in `C.main()` binds to `B.foo()` instead of `A.foo()` as it did in $\mathcal{H}_1$. Hence, behavior of $\mathcal{H}_1$'s client `C.main()` is not preserved.

Figure 6 shows an overriding composition (constructed using $\langle \mathcal{H}_1, \texttt{A.foo()} \rangle \ll \langle \mathcal{H}_2, \texttt{A.foo()} \rangle$) that is free of overriding static interference. However, in the composed hierarchy, variable `x` is bound to a `B`-object instead of an `A`-object. Thus, the call `x.bar()` suddenly resolves to `B.bar()` instead of `A.bar()` as in $\mathcal{H}_1$.

3.2 Dynamic interference

We will use the term *dynamic interference* to refer to run-time behavioral changes such as the ones in the above examples. Some additional definitions are required to make this notion precise. To this end, we will use an operational semantics, where the effect of statement execution is described as a state transformation.

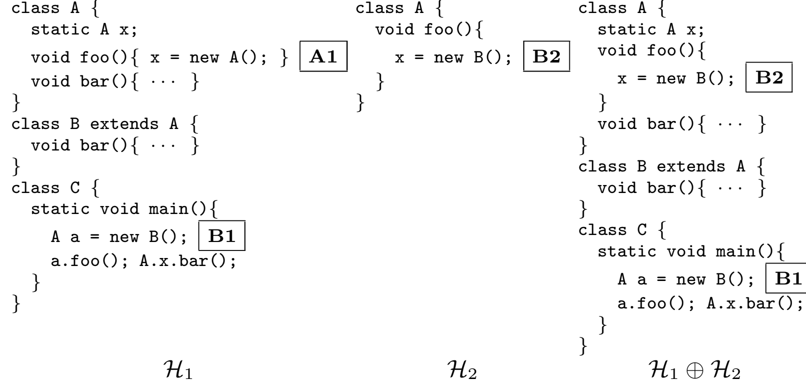

Fig. 6. Dynamic interference in an overriding composition.

Definition 8 (state, state transformation).

1. *A program state maps variables to values:* $\sigma \in \Sigma = Var \rightarrow Value$.
2. *The effect of executing a statement* S *is a state transformation:* $\sigma \xrightarrow{S} \sigma'$.

Details of *Var* and *Value* are left unspecified, as they are not important for our purposes. The reader may consult [7, 13] for complete operational semantics of relevant Java subsets. *Var* includes local variables, parameters, and `this`-pointers in method invocation stack frames—note that the domain of *Var* may change as execution proceeds. *Value* comprises primitive values (e.g., integers) and objects in the heap. In order to model reference-typed fields, we assume that *Var* also contains an element for each field f of an object o, where o is an object in *Value* whose type contains a field f (either directly or through inheritance).

Now let us assume that we have a hierarchy $\mathcal{H}$ together with some client code K which is type correct with respect to $\mathcal{H}$. We define:

Definition 9 (execution sequence).

1. *An execution sequence of a hierarchy* $\mathcal{H}$ *is the (finite or infinite) sequence of statements* $E(\mathcal{H}, K, I, \sigma_0) = S_1, S_2, S_3, \ldots$ *which results from executing the client code* K *of* $\mathcal{H}$ *with input* I *in initial state* σ_0*. The corresponding sequence of program states is* $\Sigma(\mathcal{H}, K, I, \sigma_0) = \sigma_0 \rightarrow \sigma_1 \rightarrow \sigma_2 \rightarrow \sigma_3 \ldots$.
2. *The statement subsequence of* $S_1, S_2 \ldots$ *consisting only of member accesses (data member accesses or method calls) is denoted* $M(\mathcal{H}, K, I, \sigma_0) = S_{\nu_1}, S_{\nu_2}, \ldots$ *where* $S_{\nu_i} = S_j \in E(\mathcal{H}, K, I, \sigma_0)$*. The corresponding sequence of invoked target methods is denoted* $T(\mathcal{H}, K, I, \sigma_0) = t_{\nu_1}, t_{\nu_2}, \ldots$ *where each* t_{ν_i} *is the method that is actually invoked at run-time.*

Definition 10 states that two hierarchies $\mathcal{H}$ and $\mathcal{H}'$ are behaviorally equivalent if the same sequence of statements is executed using the two hierarchies, for all given clients of $\mathcal{H}$ with appropriate inputs and initial states. Definition 11 states that a composed hierarchy $\mathcal{H}_1 \oplus \mathcal{H}_2$ exhibits *dynamic interference* if $\mathcal{H}_1$ and $\mathcal{H}_1 \oplus \mathcal{H}_2$ are not behaviorally equivalent (for some client of $\mathcal{H}_1$ with associated input and initial state), or if $\mathcal{H}_2$ and $\mathcal{H}_1 \oplus \mathcal{H}_2$ are not behaviorally equivalent (for some client of $\mathcal{H}_2$ with associated input and initial state).

Definition 10 (behavioral equivalence). *Two hierarchies $\mathcal{H}$, $\mathcal{H}'$ are behaviorally equivalent iff for all clients K of $\mathcal{H}$ with appropriate inputs and initial states I, σ_0 we have that $E(\mathcal{H}, K, I, \sigma_0) = E(\mathcal{H}', K, I, \sigma_0)$.*

Definition 11 (dynamic interference). *$\mathcal{H}_1 \oplus \mathcal{H}_2$ contains dynamic interference, if (for some $\mathcal{H}_1$-client K with associated I, σ_0) $\mathcal{H}_1$ and $\mathcal{H}_1 \oplus \mathcal{H}_2$ are not behaviorally equivalent, or if (for some $\mathcal{H}_2$-client K with associated I, σ_0) $\mathcal{H}_2$ and $\mathcal{H}_1 \oplus \mathcal{H}_2$ are not behaviorally equivalent.*

Remark. From an observational point of view, $E(\mathcal{H}_1, K, I, \sigma_0) = E(\mathcal{H}_1 \oplus \mathcal{H}_2, K, I, \sigma_0)$ is not a necessary condition for behavioral equivalence, because a modified sequence of statements might still produce the same visible effects. However, we are not interested in cases where the observable behavior of a client of a composed hierarchy is *accidentally* identical to its original behavior.

3.3 Checking for dynamic interference

We would like to *verify* whether or not a certain composition exhibits dynamic interference. In general, determining whether or not two arbitrary programs will execute the same statement sequences for all possible inputs is undecidable of course. However, for the compositions studied in this paper, the situation is not hopeless. Our approach will be to develop a *noninterference criterion* that implies $E(\mathcal{H}_1, K, I, \sigma_0) = E(\mathcal{H}_1 \oplus \mathcal{H}_2, K, I, \sigma_0)$ that is based on static analysis information. This approach is also used in our earlier work on semantics-preserving class hierarchy transformations [17, 18]. Being a sufficient, but not a necessary condition, our criterion may occasionally generate false alarms. However, we believe that the impact analysis of Section 4 will provide the user with sufficient information to determine whether reported interferences can occur in practice.

Definition 12 defines a function *srcHierarchy* that defines the class hierarchy that a program construct originated from. The *srcHierarchy* of an object is defined as the *srcHierarchy* of the statement that created it. Definition 13 uses *srcHierarchy* to define the *projection* of a state onto a hierarchy.

Definition 12 (srcHierarchy). *For $\mathcal{H} = (\mathcal{C}, \leq)$, $C = \langle c, M \rangle \in \mathcal{H}$, $m \in M$, and a statement s in $body(m)$, we write $srcHierarchy(C) = srcHierarchy(m) = srcHierarchy(s) = \mathcal{H}$. Moreover, let $s \equiv \texttt{new}\ C(\cdots)$ be an object creation site, and let object $o \in Value$ be an instance of C created by s at run-time. Then, $srcHierarchy(o) = srcHierarchy(s)$. Further, for any $x \in Var$, we define $srcHierarchy(x) = srcHierarchy(s)$, where s is the static program part responsible for the creation of x at run-time.*

Definition 13 (state projection). *The projection of a program state σ onto a hierarchy $\mathcal{H}$ is defined as $\sigma|\mathcal{H} = \{x \mapsto v \mid x \mapsto v \in \sigma, srcHierarchy(x) = \mathcal{H}\}$. Moreover, we extend the projection operator to apply to a sequence of program states as follows: $\Sigma(\mathcal{H}_1 \oplus \mathcal{H}_2, K, I, \sigma_0)|\mathcal{H}_1 = \sigma_0|\mathcal{H}_1 \rightarrow \sigma_1|\mathcal{H}_1 \rightarrow \sigma_2|\mathcal{H}_1 \rightarrow \cdots$, where $\Sigma(\mathcal{H}_1 \oplus \mathcal{H}_2, K, I, \sigma_0) = \sigma_0 \rightarrow \sigma_1 \rightarrow \sigma_2 \rightarrow \cdots$.*

The noninterference criterion relies on information produced by a *points-to* analysis [1]. A points-to analysis computes for each reference-typed variable the set of objects that it may point to. Definitions 14 and 15 define appropriate notions *ObjectRefs* of object references and *Objects* of objects.

Definition 14 (object reference). *Let $\mathcal{H}$ be a hierarchy. Then, $\mathit{ObjectRefs}(\mathcal{H}) \subseteq \mathit{Var}$ is the set of all object references in $\mathcal{H}$. This includes class-typed local variables, method parameters, fields, static variables, and* `this` *pointers of methods.*

Definition 15 (object). *Let $\mathcal{H}$ be a hierarchy. Then, $\mathit{Objects}(\mathcal{H})$ is the set of all object creation sites in $\mathcal{H}$, that is, all statements $S \equiv new\ C(...)$; occurring in some method body. Moreover, for $o \in \mathit{Objects}(\mathcal{H})$, $o \equiv new\ C(...)$, we define $\mathit{TypeOf}(\mathcal{H}, o) = C$.*

Object creation sites in $\mathit{Objects}(\mathcal{H})$ should not be confused with run-time objects in *Value*. Finally, Definition 16 formalizes the notion of points-to sets. We do not make any assumptions about the specific algorithm used to compute these points-to sets. Any method suitable for object-oriented languages will do (e.g., [11, 14]).

Definition 16 (points-to sets). *Let $\mathcal{H}$ be a hierarchy, and let $p \in \mathit{ObjectRefs}(\mathcal{H})$. Then, $\mathit{PointsTo}(\mathcal{H}, p) \subseteq \mathit{Objects}(\mathcal{H})$ is the set of all objects (represented by creation sites) that object reference p might point to at run-time.*

The noninterference criterion (Definition 17) states that the method invoked by a virtual method call $p.m(\cdots)$ (as determined by applying *StaticLookup* on the receiver object) in $\mathcal{H}_1 \oplus \mathcal{H}_2$ is the same as it was in $\mathcal{H}_1$. Note that the condition does not say anything about $\mathcal{H}_2$ objects. While the points-to sets in the composed hierarchy may also contain $\mathcal{H}_2$ objects, these objects are never created by clients of $\mathcal{H}_1$, and therefore need not be considered.

Definition 17 (noninterference criterion).
A composition $\mathcal{H}_1 \oplus \mathcal{H}_2$ meets the noninterference criterion if for all $p \in \mathit{ObjectRefs}(\mathcal{H}_1)$, for all method calls $p.m(\cdots)$ in $\mathcal{H}_1$, and for all $o \in \mathit{PointsTo}(\mathcal{H}_1 \oplus \mathcal{H}_2, p) \cap \mathit{Objects}(\mathcal{H}_1)$ we have that $\mathit{StaticLookup}(\mathcal{H}_1, T, m) = \mathit{StaticLookup}(\mathcal{H}_1 \oplus \mathcal{H}_2, T', m)$ where $T = \mathit{TypeOf}(\mathcal{H}_1, o)$, and $T' = \mathit{TypeOf}(\mathcal{H}_1 \oplus \mathcal{H}_2, o)$.

The use of points-to information deserves a few more comments. First, one might wonder about the consequences of using imprecise points-to information (i.e., overly large points-to sets). In this case, the "for all o" in the noninterference criterion runs over a larger scope than necessary, making the criterion stronger than necessary, and spurious interference may be reported. However, the criterion is safe in the sense that it will never erroneously report non-interference.

Note also that the criterion is much more precise than the static checks from Section 2. To see this, consider again the examples of Figures 5 and 6. One could argue that the simple interferences in these examples are not really dynamic. In

fact, one could report *static interference* in the examples of of Figures 5 and 6 by modifying $members(C)$ to also include members in superclasses of C (as opposed to only the members defined locally in C). So why use the more complex criterion of Definition 17? The reason is that, for large programs, the suggested modified static interference check will report interference even if the class exhibiting the changed behavior is never used in the program. In Definition 17, the scope of the key condition

$$StaticLookup(\mathcal{H}_1 \oplus \mathcal{H}_2, T, m) = StaticLookup(\mathcal{H}_1, T', m)$$

is limited by the size of the points-to set associated with the receiver of the method call. This effectively restricts the condition to method calls that are actually reachable, and to many fewer calling relationships than those that are possible, resulting in many fewer spurious interferences being reported.

Most points-to analysis algorithms compute information that is valid for a specific client K. This has the advantage that the points-to sets are more precise, reducing the number of false alarms. However, in this case the noninterference criterion only holds for the client K. To compute results that are safe for *any* client, one could employ algorithms such as [15] that are capable of analyzing incomplete programs.

Concerning the complexity of the interference test, it is dominated by the computation of points-to information, as the test itself just performs two static lookups per member access, which is linear in the program size. Various points-to algorithms of various precision are known, ranging from Steensgaard's almost linear algorithm (which scales to millions of LOC) to Andersen' cubic algorithm which has recently been scaled to a million-line C-program [6] by using a new approach for dynamically computing transitive closures. The performance of this algorithm on object-oriented applications is still unknown, as far as we know.

3.4 Justification

We will now demonstrate that the noninterference criterion of Definition 17 ensures that the behavior of client K of $\mathcal{H}_1$ is preserved. The analogous argument for $\mathcal{H}_2$ is completely symmetrical.

Lemma 3 states that it is sufficient to demonstrate that the sequence of call targets in $\mathcal{H}_1$ does not change after composition.

Lemma 3. *For all K, I and σ_0, we have that:* $T(\mathcal{H}_1, K, I, \sigma_0) = T(\mathcal{H}_1 \oplus \mathcal{H}_2, K, I, \sigma_0) \implies E(\mathcal{H}_1, K, I, \sigma_0) = E(\mathcal{H}_1 \oplus \mathcal{H}_2, K, I, \sigma_0)$ *and therefore:* $T(\mathcal{H}_1, K, I, \sigma_0) = T(\mathcal{H}_1 \oplus \mathcal{H}_2, K, I, \sigma_0) \implies \Sigma(\mathcal{H}_1, K, I, \sigma_0) = \Sigma(\mathcal{H}_1 \oplus \mathcal{H}_2, K, I, \sigma_0)|\mathcal{H}_1$

Proof. The proof is by induction on the length n of sequence $E(\mathcal{H}_1, K, I, \sigma_0)$. For $n = 0$, the statement is trivial, as both hierarchies start in state σ_0. Now consider statement S_n in $E(\mathcal{H}_1, K, I, \sigma_0)$. By induction, the previously executed statements $S_1, S_2, \ldots S_{n-1}$ are the same in both $E(\mathcal{H}_1, K, I, \sigma_0)$ and $E(\mathcal{H}_1 \oplus \mathcal{H}_2, K, I, \sigma_0)$, and lead to corresponding states σ_{n-1} and $\sigma_{n-1}|\mathcal{H}_1$, respectively.

Now, S_n is a statement at some position π in some method m_1 of $\mathcal{H}_1$. By assumption, the sequence of executed method bodies is the same in $\mathcal{H}_1 \oplus \mathcal{H}_2$. Since there is no static interference, we may conclude that we are at the same position π in some method m_2 in $\mathcal{H}_1 \oplus \mathcal{H}_2$, for which $\mathit{body}(m_1) = \mathit{body}(m_2)$. Hence, S_n is also the next statement to be executed in $E(\mathcal{H}_1 \oplus \mathcal{H}_2, K, I, \sigma_0)$.

Now there are two cases. 1) S_n is a method call $p.m(\cdots)$. Then, by assumption, this call resolves to the same target t_{ν_i} which is executed in the same state σ_{n-1}. Thus, after execution of the method body, both hierarchies are in the same state σ_n at the same position π' in the same method bodies m_1 and m_2. 2) S_n is not a method call. Then, it must be the same in m_1 and m_2 due to the absence of static interference. Hence, after execution, the same state σ_n is reached. □

Theorem 1 (correctness of criterion). *Let $\mathcal{H}_1 \oplus \mathcal{H}_2$ be a composition that meets the noninterference criterion of Definition 17. Then, we have that: $E(\mathcal{H}_1, K, I, \sigma_0) = E(\mathcal{H}_1 \oplus \mathcal{H}_2, K, I, \sigma_0)$ and $\Sigma(\mathcal{H}_1, K, I, \sigma_0) = \Sigma(\mathcal{H}_1 \oplus \mathcal{H}_2, K, I, \sigma_0)|\mathcal{H}_1$, for all K, I, σ_0.*

Proof. Again the proof is by induction on n. For $n = 0$ the statement is trivial, as both hierarchies start in state σ_0. Now consider statement S_n in $E(\mathcal{H}_1, K, I, \sigma_0)$. By induction, the previously executed statements $S_1, S_2, \ldots S_{n-1}$ are the same in both $E(\mathcal{H}_1, K, I, \sigma_0)$ and $E(\mathcal{H}_1 \oplus \mathcal{H}_2, K, I, \sigma_0)$ and lead to corresponding states σ_{n-1} and $\sigma_{n-1}|\mathcal{H}_1$, respectively. As in the previous lemma, we may conclude that S_n is the same in both execution sequences. (This time the necessary fact that the previous call targets have been the same is not by assumption, but by induction: if $S_1, S_2, \ldots$ are the same in both $E(\mathcal{H}_1, K, I, \sigma_0)$ and $E(\mathcal{H}_1 \oplus \mathcal{H}_2, K, I, \sigma_0)$, so are $S_{\nu_1}, S_{\nu_2}, \ldots$ and $t_{\nu_1}, t_{\nu_2}, \ldots$).

Now there are two cases. If S_n is not a method call or data member access, we may conclude that the corresponding states σ_n and $\sigma_n|\mathcal{H}_1$ are produced in $\Sigma(\mathcal{H}_1, K, I, \sigma_0)$ and $\Sigma(\mathcal{H}_1 \oplus \mathcal{H}_2, K, I, \sigma_0)|\mathcal{H}_1$, respectively. In case $S_n = p.m(x)$, we know $\sigma_{n-1}(p) \in \mathit{PointsTo}(\mathcal{H}_1 \oplus \mathcal{H}_2, p)$ (remember that σ_{n-1} was reached in both $\mathcal{H}_1$ and $\mathcal{H}_1 \oplus \mathcal{H}_2$). Furthermore, $\sigma_{n-1}(p) \in \mathit{Objects}(\mathcal{H}_1)$, as $\sigma_{n-1}|\mathcal{H}_1 \in \Sigma(\mathcal{H}_1, K, I, \sigma_0)$, and $\mathcal{H}_1$ does not contain $\mathcal{H}_2$ objects. We know $\mathit{StaticLookup}(\mathcal{H}_1, \mathit{TypeOf}(\mathcal{H}_1, o), m) \neq \bot$ and by the type correctness lemma thus $\mathit{StaticLookup}(\mathcal{H}_1 \oplus \mathcal{H}_2, \mathit{TypeOf}(\mathcal{H}_1 \oplus \mathcal{H}_2, o), m) \neq \bot$. By the static noninterference criterion, both static lookups must compute the same result, namely method definition (resp. data member) m. After execution of m's body (resp. access to m's value in state σ_{n-1}), we may as in the above lemma conclude that execution of both hierarchies is in the same state σ_n. □

Example 6. For the example of Figure 5, we have $\mathit{PointsTo}(\mathcal{H}_1, \texttt{o}) = \mathit{PointsTo}(\mathcal{H}_1 \oplus \mathcal{H}_2, \texttt{o}) = \{\texttt{B1}\}$. For the method call `o.foo()`, we obtain $\mathit{StaticLookup}(\mathcal{H}_1, \mathit{TypeOf}(\mathcal{H}_1, \texttt{B1}), \texttt{foo}()) = \mathit{StaticLookup}(\mathcal{H}_1, \texttt{B}, \texttt{foo}()) = \texttt{A.foo}()$, but $\mathit{StaticLookup}(\mathcal{H}_1 \oplus \mathcal{H}_2, \mathit{TypeOf}(\mathcal{H}_1 \oplus \mathcal{H}_2, \texttt{B1}), \texttt{foo}()) = \mathit{StaticLookup}(\mathcal{H}_1 \oplus \mathcal{H}_2, \texttt{B}, \texttt{foo}()) = \texttt{B.foo}()$. Hence, dynamic interference is reported. □

In Example 7 and in subsequent examples, we use the labels shown in boxes in Figure 1 to identify object creation sites.

Example 7. For client `Driver1.main()` in Figure 2, we obtain $\mathit{PointsTo}(\mathcal{H}_1 \oplus \mathcal{H}_2, \mathtt{c1}) \cap \mathit{Objects}(\mathcal{H}_1) = \{ \mathtt{C1} \}$, $\mathit{PointsTo}(\mathcal{H}_1 \oplus \mathcal{H}_2, \mathtt{p1}) \cap \mathit{Objects}(\mathcal{H}_1) = \{ \mathtt{P1} \}$. Hence, $\mathit{StaticLookup}(\mathcal{H}_1, \mathit{TypeOf}(\mathcal{H}_1, \mathtt{C1}), \mathtt{enroll}()) = \mathtt{Course.enroll}() = \mathit{StaticLookup}(\mathcal{H}_1 \oplus \mathcal{H}_2, \mathit{TypeOf}(\mathcal{H}_1 \oplus \mathcal{H}_2, \mathtt{C1}), \mathtt{enroll}())$. Moreover, we have that $\mathit{StaticLookup}(\mathcal{H}_1, \mathit{TypeOf}(\mathcal{H}_1, \mathtt{C1}), \mathtt{assign}()) = \mathtt{Course.assign}() = \mathit{StaticLookup}(\mathcal{H}_1 \oplus \mathcal{H}_2, \mathit{TypeOf}(\mathcal{H}_1 \oplus \mathcal{H}_2, \mathtt{C1}), \mathtt{assign}())$, and that $\mathit{StaticLookup}(\mathcal{H}_1, \mathit{TypeOf}(\mathcal{H}_1, \mathtt{P1}), \mathtt{approveGraduation}()) = \mathtt{Professor.approveGraduation}() = \mathit{StaticLookup}(\mathcal{H}_1 \oplus \mathcal{H}_2, \mathit{TypeOf}(\mathcal{H}_1 \oplus \mathcal{H}_2, \mathtt{P1}),$ `Professor.approveGraduation()`). Similar arguments can be made for all other method calls in Figure 2. Hence, the behavior of `Driver1.main()` in $\mathcal{H}_1$ is preserved in $\mathcal{H}_1 \oplus \mathcal{H}_2$. □

In fact, the basic composition in Figure 2 preserves the behavior of *any* possible client. Potential clients may introduce arbitrary allocation sites and arbitrary points-to relationships. A conservative approximation must therefore assume that for any member access $p.m(\cdots) \in \mathcal{H}_1$, $\mathit{PointsTo}(\mathcal{H}_1 \oplus \mathcal{H}_2, p) \cap \mathit{Objects}(\mathcal{H}_1)$ contains all allocation sites $S \in \mathit{Objects}(\mathcal{H}_1)$ where $\mathit{TypeOf}(\mathcal{H}_1 \oplus \mathcal{H}_2, S) \leq \mathit{TypeOf}(\mathcal{H}_1 \oplus \mathcal{H}_2, p)$. Nevertheless, $\mathit{StaticLookup}(\mathcal{H}_1, \mathit{TypeOf}(\mathcal{H}_1, S), m) = \mathit{StaticLookup}(\mathcal{H}_1 \oplus \mathcal{H}_2, \mathit{TypeOf}(\mathcal{H}_1 \oplus \mathcal{H}_2, S), m)$, because in the example the methods from $\mathcal{H}_1$ and $\mathcal{H}_2$ are either disjoint or identical.

3.5 Overriding compositions

The noninterference criterion was designed for basic compositions. It can be applied to overriding compositions as well, but will report failure as soon as a method call $p.m(\cdots)$ resolves to methods that have different bodies in $\mathcal{H}_1$ and $\mathcal{H}_1 \oplus \mathcal{H}_2$. Nevertheless, interference checks may still succeed if the overridden methods are not reachable from client code. Constructing an interference check for overriding compositions that ignores conflicts that users are aware of (via the $\ll$-ordering) is a topic for future research.

Example 8. Let us apply the criterion to Figure 6. We have $\mathit{PointsTo}(\mathcal{H}_1, \mathtt{a}) = \mathit{PointsTo}(\mathcal{H}_1 \oplus \mathcal{H}_2, \mathtt{a}) = \{ \mathtt{B1} \}$, but due to the overriding, $\mathit{StaticLookup}(\mathcal{H}_1, \mathit{TypeOf}(\mathcal{H}_1, \mathtt{B1}), \mathtt{foo}()) = \mathtt{A.foo}()_{\mathcal{H}_1} \neq \mathtt{A.foo}()_{\mathcal{H}_2} = \mathit{StaticLookup}(\mathcal{H}_1 \oplus \mathcal{H}_2, \mathit{TypeOf}(\mathcal{H}_1 \oplus \mathcal{H}_2, \mathtt{B1}), \mathtt{foo}())$. Here, we use subscripts $\mathcal{H}_1$ and $\mathcal{H}_2$ to indicate the hierarchies that the different methods `A.foo()` originate from. Hence, the behavior of call `a.foo()` is not preserved. For call `x.bar()`, we obtain $\mathit{PointsTo}(\mathcal{H}_1 \oplus \mathcal{H}_2, \mathtt{x}) = \{ \mathtt{B2} \}$, hence $\mathit{PointsTo}(\mathcal{H}_1 \oplus \mathcal{H}_2, \mathtt{x}) \cap \mathit{Objects}(\mathcal{H}_1) = \emptyset$. Thus the criterion is trivially satisfied for `x.bar()`. □

Remark: In Example 8, one might wonder why the criterion is satisfied for `x.bar()` despite the fact that the behavior of this call obviously changed: We have that `B.bar()` is called in $\mathcal{H}_1 \oplus \mathcal{H}_2$ whereas `A.bar()` was called in $\mathcal{H}_1$. This *secondary* behavioral change is caused by another change in behavior (in this case, the changed behavior of call `a.foo()`, which causes `x` to be bound

to an object of type `B`). While the noninterference criterion does not necessarily detect secondary behavioral changes, it *does* find all primary changes, which suffices to guarantee behavioral equivalence. We plan to use impact analysis to obtain a more precise understanding of where behavioral changes occur, as will be discussed in Section 4.

4 Impact Analysis

When the purpose of composition is to add functionality that is not completely orthogonal to the system's existing functionality, changes in behavior are often unavoidable. The interference check of Section 3 determines whether the behavior of a specific client K is affected. In principle, one could manually apply the interference check to successively smaller clients to determine the impact of the composition. However, such a non-automated process is tedious and labor-intensive, and problematic in cases where it is not possible to partition the code in K. In order to provide a less labor-intensive approach, we plan to adapt the change impact analysis of [16] in order to automatically determine the set of program constructs affected by a composition. The remainder of this section presents some preliminary ideas on how this can be accomplished.

4.1 Change impact analysis

We begin with a brief review of the change impact analysis of [16]. In this work, it is assumed that a program is covered by a set of regression test drivers $\mathcal{T} = t_1, \cdots, t_n$. Each test driver is assumed to consist of a separate `main` routine from which methods in the application are called. Prior to the editing session, a *call graph* G_i is built for each t_i. Any of several existing call graph construction algorithms can be used. We plan to employ an algorithm that has been demonstrated to scale to large applications [21].

After the user has ended the editing session, the edits are decomposed into a set $\mathcal{A}$ of atomic changes. $\mathcal{A}$ consists of (i) a set **AC** of (empty) classes that have been added, (ii) a set **DC** of classes that have been deleted, (iii) a set **AM** of methods that have been added, (iv) a set **DM** of methods that have been deleted, (v) a set **CM** of methods whose body contains one or more changed statements, (vi) a set **AF** of fields that have been added, (vii) a set **DF** of fields that have been deleted, and (viii) a set **LC** of elements of the form $\langle T, C.m() \rangle$, indicating that the dispatch behavior for a call to $C.m()$ on an object of type T has changed. In general, a simple user edit can imply several atomic changes. For example, addition of a method may involve the addition of an empty method (in **AM**), a change in dispatch behavior (in **LC**) of existing call sites in cases where the added method overrides an existing method, and a change of a method body (in **CM**).

By analyzing the call graphs G_i and the set of atomic changes $\mathcal{A}$, a subset of *affected tests* $\mathcal{T}' \subseteq \mathcal{T}$ is determined. Figure 7 shows how correlating the nodes and edges in the call graphs for the test drivers with the **CM**, **DM**, and **LC**

$$
\begin{aligned}
\mathit{AffectedTests}(\mathcal{T},\mathcal{A}) = \; & \{\, t_i \mid t_i \in \mathcal{T},\; \mathit{Nodes}(P,t_i) \cap (\mathbf{CM} \cup \mathbf{DM})) \neq \emptyset \,\} \cup \\
& \{\, t_i \mid t_i \in \mathcal{T},\; n \in \mathit{Nodes}(P,t_i),\; n \rightarrow_B A.m \in \mathit{Edges}(P,t_i), \\
& \quad \langle B, X.m() \rangle \in \mathbf{LC},\; B <^* A \leq^* X \,\}
\end{aligned}
$$

Fig. 7. Definition of *AffectedTests* (taken from [16]).

changes leads to the identification of a set of affected tests t_i. Here, $\mathit{Nodes}(P, t_i)$ and $\mathit{Edges}(P, t_i)$ denote the set of nodes resp. edges in the call graph for test driver t_i. Informally, the formula shown in the figure states that a test driver t_i is potentially affected if a node in the call graph for t_i corresponds to a deleted or changed method (line 1), or if one of the edges in the call graph for t_i corresponds to a dispatch relation that has been changed (lines 2 3).

Any test driver that is not in $\mathcal{T}'$ is guaranteed to have the same behavior as before. If a test driver t_i occurs in $\mathcal{T}$, the user can run this test to determine if the new behavior of the test meets his expectations. If this is not the case, an additional analysis (also based on call graphs) determines a *subset* of atomic changes $\mathcal{A}' \subseteq \mathcal{A}$ that contribute to t_i's altered behavior. The user can apply successively larger subsets of $\mathcal{A}'$ to identify the change that "broke" test driver t_i. Alternatively, divide-and-conquer strategies similar to the ones in [23] can be applied to quickly narrow down the search space.

4.2 Impact analysis for composition

We plan to adapt the analysis of [16] to determine the impact of a composition $\mathcal{H}_1 \oplus \mathcal{H}_2$ on a set of test drivers $\mathcal{T} = t_1, \cdots, t_k$ associated with hierarchy $\mathcal{H}_i$ by interpreting a composition as a set of changes w.r.t $\mathcal{H}_i$. To do so, we need to establish a relationship between classes/members in $\mathcal{H}_1 \oplus \mathcal{H}_2$, and the classes/members in $\mathcal{H}_1$ and $\mathcal{H}_2$ that these classes/members originate from. This is expressed by Definition 18 below.

Definition 18 (origin). *Let $\mathcal{H}_1 = (\mathcal{C}_1, \leq_1)$ and $\mathcal{H}_2 = (\mathcal{C}_2, \leq_2)$ be two class hierarchies, let $\mathcal{H} = \mathcal{H}_1 \oplus \mathcal{H}_2 = (\mathcal{C}, \leq)$, and let $\mathcal{E}$ be the set of equivalence classes constructed by Definition 2. Furthermore, let C be a class in $\mathcal{C}$, and let $m \in \mathrm{members}(C)$. Define:*

$$
\begin{aligned}
\mathit{origin}(C) &= \{\, \langle C', \mathcal{H} \rangle \mid S \in \mathcal{E}, \mathrm{name}(S) = \mathrm{name}(C),\; \langle C', \mathcal{H} \rangle \in S \,\} \\
\mathit{origin}(m) &= \{\, \langle C', m', \mathcal{H} \rangle \mid \langle C', \mathcal{H} \rangle \in \mathit{origin}(C),\; m' \in \mathrm{members}(C'), \\
& \qquad \mathit{namesig}(m) = \mathit{namesig}(m'),\; \mathit{body}(m) = \mathit{body}(m')
\end{aligned}
$$

Definition 18 defines the origin of a class C in $\mathcal{C}$ as the set of (class,hierarchy) pairs in its equivalence class, and the origin of a method m in C is the set of methods in each such class with the same name and body as m. Note that these definitions work with both of the *members* definitions given earlier.

Example 9. For hierarchy $\mathcal{H} = \mathcal{H}_1 \oplus \mathcal{H}_3$ of Figure 3 we have $\mathit{origin}(\texttt{Professor}) = \{\, \langle \texttt{Professor}, \mathcal{H}_1 \rangle,\; \langle \texttt{Professor}, \mathcal{H}_3 \rangle \,\}$ and $\mathit{origin}(\texttt{Professor.approveGraduation}) = \{\, \langle \texttt{Professor}, \texttt{approveGraduation}, \mathcal{H}_3 \rangle \,\}$. □

It is now straightforward to construct the sets of atomic changes w.r.t. one of the original hierarchies. For example, for the composition $\mathcal{H} = \mathcal{H}_1 \oplus \mathcal{H}_3$, where $\mathcal{H} = \langle \mathcal{C}, \leq \rangle$, $\mathcal{H}_1 = \langle \mathcal{C}_1, \leq_1 \rangle$, and $\mathcal{H}_3 = \langle \mathcal{C}_4, \leq_4 \rangle$, the sets **AC** and **CM** w.r.t. $\mathcal{H}_1$ may be computed as:

$$
\begin{aligned}
AC &= \{\, C' \mid \langle C', \mathcal{H}' \rangle \in \mathit{origin}(C), C \in \mathcal{C}, \mathcal{H}' \neq \mathcal{H}_1 \,\} \\
CM &= \{\, m' \mid \langle C', \mathcal{H}_1 \rangle \in \mathit{origin}(C), C \in \mathcal{C}, m' \in \mathit{members}(C), \\
&\qquad \mathit{origin}(m') = \emptyset,\ \nexists m \text{ in } \mathit{members}(\text{C'}) \text{ s.t.} \\
&\qquad \mathit{namesig}(m) = \mathit{namesig}(m') \text{ and } \mathit{body}(m) = \mathit{body}(m') \,\}
\end{aligned}
$$

The other sets of atomic changes are computed similarly. We can now apply the analysis of [16] to determine the impact of the composition.

Example 10. Consider the overriding composition of Figure 3, for which we previously found that behavior could not be preserved (see Example 8). We will now apply impact analysis to obtain a more precise understanding of where the interferences occur. Interpreting $\mathcal{H}_1 \oplus \mathcal{H}_3$ as a set of changes w.r.t. $\mathcal{H}_1$, we find that `Driver1.main()` is affected, because **CM** contains method `Professor.approveGraduation()`, which occurs in the call graph for `Driver1.main()`. Moreover, interpreting $\mathcal{H}_1 \oplus \mathcal{H}_3$ as a set of changes w.r.t. $\mathcal{H}_3$, we find that `Driver3.main()` is not affected, because the call graph for `Driver3.main()` does not contain any added, changed, or deleted methods, or any edges corresponding to a changed dispatch behavior.

Thus far, we have computed the impact of a composition on a set of test drivers. In order to obtain more fine-grained information, one could construct a separate call graph for each method m in hierarchy $\mathcal{H}_1$ (using appropriate conservative assumptions about the run-time types of parameters and accessed fields), and proceed as before. Then, impact could be reported as the set of methods whose behavior might have changed. For example, if separate call graphs are constructed for all methods of $\mathcal{H}_1$ in the overriding composition $\mathcal{H} \oplus \mathcal{H}_3$ of Figure 3, we can report that only the behavior of methods `Driver1.main()` and `Professor.approveGraduation()` is impacted by the composition because these are the only methods that transitively call methods whose behavior may have changed. Space limitations prevent us from providing more details.

5 Related work

Research on aspect-oriented software development has been gaining in popularity recently [24]. In essence, the goal of this field is to obtain more extensible and reusable designs by distributing unrelated functionality over disjoint hierarchies or aspects. To achieve this, a mechanism for *composing* these functionalities into executable code is needed. In our setting, this is accomplished by composing hierarchies. Aspect-oriented languages such as AspectJ [8] have language constructs that allow one to specify the conditions under which a piece of *advice* is "woven in" at a *joint point*. Until recently, there has been very little work on the semantic foundations of composition.

The work most closely related to ours is the Aspect Sandbox (ASB) project by Wand et al. [22], who incorporate several key aspect-oriented language constructs such as join points, pointcut designators, and advice into a simple language with procedures, classes, and objects. Wand et al. formalize the semantics of this language using a denotational semantices. Wand et al. do not provide any guarantees about the noninterference of aspects, nor do they determine the semantic impact of "weaving in" an aspect.

Ernst presented a class hierarchy composition system, where the composition operator is built into the syntax of the programming language gbeta [4]. Explicitely specified compositions may trigger propagation of implicit compositions, for example an explicit combination of methods or classes may trigger implicit composition of other (e.g. auxiliary) methods or classes. Ernst showed that his composition system and propagation mechanism preserve static type correctness, but nothing is said about preservation of dynamic client behaviour.

Our approach is similar in spirit to the work by Binkley et al. [2] on the integration of C programs that are variations of a common base, by analyzing and merging their program dependence graphs. Similar to our work, Binkley et al. use an interference test, which is a sufficient criterion for noninterference. The main differences between [2] and our work is that [2] operates at the statement level, whereas our techniques operate on calling relationships between methods. Binkley et al. do not consider object-oriented language features.

Composition of object-oriented programs has been studied by other authors in the context of component-based systems (see [12] for a collection of relevant articles). One approach is concerned with the dynamic interaction of concurrent objects, and the goal is to create new behavior as a composition of given behavior. Our approach aims to preserve old behavior while combining given behavior. Another line of research has investigated the composition of systems from components, where the components are treated as black boxes and come with some interface specification, usually in the form of a type system.

We already discussed how our impact analysis is a derivative of the change impact analysis of [16]. Offutt and Li [10, 9] also presented a change impact analysis for object-oriented programs, which only relies on structural relationships between classes and members (e.g., containment), and is therefore much less precise than approaches such as [16] that rely on static analysis.

6 Future Work

The work presented in this paper represents a first step in an ongoing effort to provide better semantic support for composition-based software development. We plan to implement the compositions and interference check of this paper, and gain practical experience with the approach. Other future work includes:

– The current paper abstracts away from peculiarities of specific programming languages. Programming languages such as Java contain several features that require further thought (e.g., exception handling).

- We plan to explore more complex compositions such as "merging" compositions, in which two interfering methods m_1 and m_2 are "merged" in some user-specified way (e.g., by constructing a new method that first executes the body of m_1 and then that of m_2).
- We plan to investigate more sophisticated interference tests can that can provide behavioral guarantees even in the presence of dynamic interference, by taking into account those conflicts that users have explicitly resolved.
- We have outlined how the impact analysis of [16] can be used to obtain a more detailed view of where behavioral interferences occur. We consider this to be a topic that needs much further thought.
- In practice, class hierarchy compositions are often performed with the intention of changing program behavior. We consider methods for distinguishing behavioral changes expected by the user from unanticipated behavioral changes to be an important research topic. Such methods could be used to filter the information produced by the impact analysis.

Acknowledgments

We are grateful to Charles Barton, Harold Ossher, and Max Störzer for comments on a draft of this paper. We would also like to thank the anonymous ECOOP referees for their detailed feedback.

References

1. L. O. Andersen. *Program Analysis and Specialization for the C Programming Language.* PhD thesis, DIKU, Univ. of Copenhagen, May 1994. DIKU report 94/19.
2. David Binkley, Susan Horwitz, and Thomas Reps. Program integration for languages with procedure calls. *ACM Transactions on Software Engineering and Methodology*, pages 3–35, January 1995.
3. B. A. Davey and H. A. Priestley. *Introduction to Lattices and Order.* Cambridge University Press, 1990.
4. Erik Ernst. Propagating class and method combination. In *Proc. European Conference on Object-Oriented programming (ECOOP99)*, pages 67–91, Lisboa, 1999.
5. William Harrison and Harold Ossher. Subject-oriented programming (a critique of pure objects). In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 411–428, 1993.
6. Nevin Heintze and Oliver Tardieu. Ultra-fast alias analysis using CLA. In *Proc. ACM SIGPLAN Symposium on Programming Language Design and Implementation (PLDI01)*, pages 254–263, Snowbird, Utah, 2001.
7. Atshushi Igarashi, Benjamin Pierce, and Philip Wadler. Featherweight Java: A minimal core calculus for Java and GJ. In *Proc. SIGPLAN/ Conference an Onject-Oriented Programmin, Systems, Languages and Applications (OOPSLA'99)*, pages 132–146. ACM, November 1999.
8. Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An overview of AspectJ. In *Proc. 15th European Conf. on Object-Oriented Programming (ECOOP'01)*, Budapest, Hungary, June 2001.

9. Michelle L. Lee. *Change Impact Analysis of Object-Oriented Software*. PhD thesis, George Mason University, 1998.
10. Michelle Lee and A. Jefferson Offutt. Algorithmic analysis of the impact of changes to object-oriented software. In *IEEE International Conference on Software Maintenance*, pages 171–184, Monterey, CA, November 1996.
11. Donglin Liang, Maikel Pennings, and Mary Jean Harrold. Extending and evaluating flow-insensitive and context-insensitive points-to analyses for Java. In *Proc. ACM SIGPLAN/SIGSOFT Workshop on on Program Analysis for Software Tools and Engineering (PASTE'01)*, pages 73–79, Snowbird, Utah, 2001.
12. Oscar Nierstrasz and Dennis Tsichritzis (ed.). *Object-Oriented Software Composition*. Prentice Hall, 1995.
13. Tobias Nipkow and David von Oheimb. Java_{light} is type safe - definitely. In *Proc. SIGPLAN/SIGACT Symposium on Principles of Program Languages (POPL'98)*, pages 161–170. ACM, January 1998.
14. Atanas Rountev, Ana Milanova, and Barbara G. Ryder. Points-to analysis for Java using annotated constraints. In *Proc. ACM SIGPLAN Conf. on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'01)*, Tampa, FL, 2001.
15. Atanas Rountev, Barbara G. Ryder, and William Landi. Data-flow analysis of program fragments. In *Proceedings of Symposium on the Foundations of Software Engineering (FSE 1999)*, pages 235–252, Toulouse, France, 1999.
16. Barbara G. Ryder and Frank Tip. Change impact analysis for object-oriented programs. In *Proceedings of the ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE'01)*, pages 46–53, Snowbird, UT, June 2001.
17. Gregor Snelting and Frank Tip. Reengineering class hierarchies using concept analysis. In *Proc. ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 99–110, Orlando, FL, November 1998.
18. G. Snelting and F. Tip. Understanding class hierarchies using concept analysis. *ACM Trans. on Programming Languages and Systems*, pages 540–582, May 2000.
19. Peri Tarr, Harold Ossher, William Harrison, and Stanley M. Sutton, Jr. N degrees of separation: Multi-dimensional separation of concerns. In *Proceedings of the 1999 International Conference on Software Engineering*, pages 107–119. IEEE Computer Society Press / ACM Press, May 1999.
20. Krishnaprasad Thirunarayan, Günter Kniesel, and Haripriyan Hampapuram. Simulating multiple inheritance and generics in Java. *Computer Languages*, 25:189–210, 1999.
21. Frank Tip and Jens Palsberg. Scalable propagation-based call graph construction algorithms. In *Proc. ACM SIGPLAN Conf. on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'00)*, pages 281–293, Minneapolis, MN, 2000. *SIGPLAN Notices* 35(10).
22. Mitchell Wand, Gregor Kiczales, and Christopher Dutchyn. A semantics for advice and dynamic join points in aspect-oriented programming. October 2001.
23. Andreas Zeller. Yesterday my program worked. Today, it does not. Why? In *Proc. of the 7th European Software Engineering Conf./7th ACM SIGSOFT Symp. on the Foundations of Software Engineering (ESEC/FSE'99)*, pages 253–267, Toulouse, France, 1999.
24. Communications of the ACM. 44(10), October 2001. Special issue on Aspect-Oriented Programming.