Generic Wrappers

Martin Büchi[1] and Wolfgang Weck[2]

[1] Turku Centre for Computer Science, Åbo Akademi University,
Lemminkäisenkatu 14A, FIN-20520 Turku,
Martin.Buechi@abo.fi
[2] Oberon microsystems Inc., Technoparkstrasse 1,
CH-8005 Zürich,
weck@oberon.ch

Abstract. Component software means reuse and separate marketing of pre-manufactured binary components. This requires components from different vendors to be composed very late, possibly by end users at run time as in compound-document frameworks.
To this aim, we propose generic wrappers, a new language construct for strongly-typed class-based languages. With generic wrappers, objects can be aggregated at run time. The aggregate belongs to a subtype of the actual type of the wrapped object. A lower bound for the type of the wrapped object is fixed at compile time. Generic wrappers are type safe and support modular reasoning.
This feature combination is required for true component software but not achieved by known wrapping and combination techniques, such as the wrapper pattern or mix-ins.
We analyze the design space for generic wrappers, e.g. overriding, forwarding vs. delegation, and snappy binding of the wrapped object. As a proof of concept, we add generic wrappers to Java and report on a mechanized type soundness proof of the latter.

1 Introduction

Component software enables the development of different parts of large software systems by separate teams, the replacement of individual software parts that evolve at different speeds without changing or reanalyzing other parts, and the marketing of independently developed building blocks. Components are binary units of independent production, acquisition, and deployment [30].

Component technology aims for late composition, possibly by the end user. Compound documents, e.g. a Word document with an embedded Excel spreadsheet and a Quicktime movie, as well as Web browser plug-ins and applets are examples of this. Late composition is a major difference between modern components and traditional subroutine libraries, such as Fortran numerical packages, which are statically linked by the developer.

Flexible late composition is one goal, prevention of unsafe compositions leading to system failures is the other. Static safety assertions are especially important for software components that are composed by third parties because systematic integration testing by the component developer is practically impossible. Where compile-time checks are

Elisa Bertino (Ed.): ECOOP 2000, LNCS 1850, pp. 201–225, 2000.
© Springer-Verlag Berlin Heidelberg 2000

impossible, as early as possible run-time detection of errors facilitates systematic testing and debugging. Type systems can make certain safety guarantees, but they often do so at the price of reduced flexibility.

In this paper we present an inflexibility problem in class-based languages and propose a new solution that partly borrows from prototype-based languages yet retains the possibility for maximal static and as-early-as-possible run-time error detection and modular reasoning.

Late composition is most pressing for items defined by different components, which may themselves be combined by an independent assembler or even by the user at run-time. Component standards such as Microsoft's COM [25], JavaBeans [29], and CORBA Components [21] are on the binary level. Components can be created in any language for which a mapping to the binary standards exists. However, binary standards are most easily programmed to in languages that support the same composition mechanisms. Furthermore, only direct language-level support can provide the desired machine checkable safety using types. Hence, composition mechanisms in programming languages are relevant, even though components are binary units.

The mechanism suggested in this paper is partly inspired by COM's aggregation, but it doesn't yet have an exact equivalent in any of the aforementioned binary component standards.

Overview. Section 2 illustrates with examples a problem of existing composition mechanisms and defines the requirements for a better solution. In Sect. 3, we show why existing technology does not sufficiently address these requirements. We introduce generic wrappers as a solution to the aforementioned problems in Sect. 4. Next, we discuss the design space for generic wrappers in Sect. 5 and the interplay with other type mechanisms in Sect. 6. As a proof of concept we add generic wrapping to Java in Sect. 7 and report on a mechanized type soundness proof of the extended language in Sect. 8. Finally, Sect. 9 points to related work and Sect. 10 draws the conclusions.

2 The Problem

In this section, we describe some applications that cannot be satisfactorily realized with existing composition mechanisms. We also introduce some terminology, and distill a set of requirements.

2.1 Examples

We illustrate the problem with examples in the realm of compound documents. Embedded views in compound documents for on-screen viewing, such as an Excel spreadsheet in a Word document, may be so large that they require their own scroll bars. Likewise, the user may want to add borders or identification tags to embedded views. It is even possible, that a user wants several such decorators added to the same embedded view.

There may exist different scroll bars from different vendors, which don't know all the other decorator or embedded view vendors. Decorators are typical examples of third-party components that users want to select to meet their specific needs. One user may want proportional scroll bars, another may like blinking borders to draw the boss'

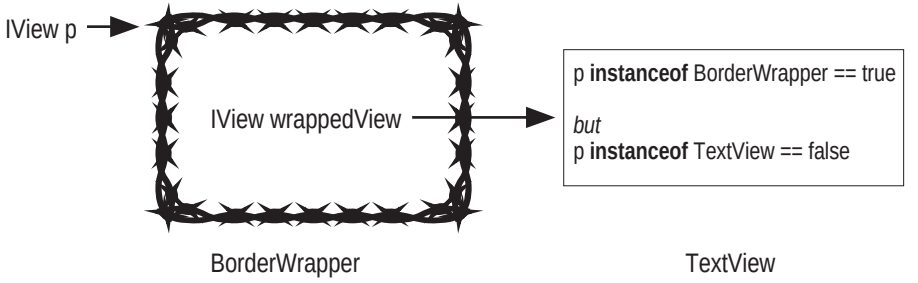


Fig. 1. The wrapper is not fully transparent to clients of the embedded view

attention to the excellent sales figures, and still another may require immutable 128-bit identification tags.

In a compound-document framework similar to Java Swing or Microsoft OLE, let IView be the interface implemented by all classes whose instances can be displayed on screen and inserted into containers. Typical examples of classes implementing IView are TextView, GraphicsView, SpreadsheetView, and ButtonView.

One way to implement decorators is with wrappers [9, Decorator Pattern]. A border wrapper is itself a view, that is it implements the IView interface. Hence it can itself be inserted into a compound document container. Furthermore the wrapper contains a reference of type IView to a wrapped view, which in a specific instance may be a TextView. The wrapper forwards most requests to the wrapped view, possibly after performing additional operations such as drawing the border.

Unfortunately, this approach has a serious disadvantage. If we wrap a border around a TextView, then the aggregate is only a BorderWrapper, but not a TextView with all of the latter's methods (Fig. 1). Hence, a spell check operation on all embedded text views in a document will not recognize a bordered TextView as containing text, unless it knows how to search inside wrappers from different manufacturers.

A standard interface, like IViewWrapper to be implemented by all view wrappers could ease the problem of searching inside different wrappers:

```
interface IViewWrapper {
  IView getWrappee();
}
```

However, instead of a simple type test, the spell checker would have to loop through all the wrappers:

```
IView q=p;
while(!(q instanceof TextView) && q instanceof IViewWrapper) {
  q=((IViewWrapper)q).getWrappee();
}
if(q instanceof TextView) {…;}
```

This solution is cumbersome for several reasons: First, it requires 5 lines of code instead of a simple type test. Second, it only works if there is a unique standard for

wrappers, such as IViewWrapper. Third, it doesn't let the wrapper maintain invariants ranging over both itself and the wrapped object because clients have direct access to the latter.

Support for certain common kinds of wrappers may also be built into the wrapped objects. For example, JComponent, the correspondence to our IView in Java Swing, supports borders as insets. However, identification tags and other kinds of wrapper that were not previewed by the Swing designers are left out.

As a second example, let us consider a forms container that requires all its embedded views to implement the interface IControl. Assume that ButtonView implements IControl and that BorderWrapper doesn't. Hence, a bordered ButtonView cannot be inserted into a forms container: The type system rightfully prevents us from passing a BorderWrapper wrapping a ButtonView as the first parameter to the method insert(IControl c, Point pos). Passing just the wrapped ButtonView as parameter to insert is not a solution, because we would loose the border. The only workaround is to change the type of the first parameter of insert to IView and test that the actual parameter implements IControl or wraps an object that does so.

2.2 Terminology

We use the following terminology: A wrapped object is called a *wrappee*. A wrapper and a wrappee together are referred to as an *aggregate*. The declared type of a variable is referred to as *static* (compile-time) *type*. The type of the actually referenced object is called the variable's *dynamic* (actual, run-time) *type*. Likewise, we distinguish between the *static* (declared, compile-time) and the *dynamic* (actual, run-time) *wrappee type*. For example, for an instance of BorderWrapper, declared to wrap an IView, and actually wrapping a TextView, the static wrappee type is IView and the dynamic wrappee type is TextView.

In discussions, we use the notation C.m to refer to the implementation of instance method m in class C. The subtype relation is taken to be reflexive; e.g., TextView is a subtype of itself.

Except where otherwise stated, the discussion in the first 6 sections applies to most strongly-typed class-based languages such as Java [10], Eiffel [17], and C++ [28]. For simplicity, we use Java terminology throughout the paper. A Java interface corresponds to a fully abstract class in Eiffel and C++.

2.3 Requirements

From the above examples we can distill a number of requirements for a wrapping mechanism. Numbers in parentheses refer to the summary of requirements in Fig. 2.

The user wants to select which border to wrap around which view. At compile time, the implementor of BorderWrapper doesn't know whether an instance of her class will wrap a TextView, a GraphicsView, or any other view that might even be only implemented in the future. Thus, the actual type and instance of the wrappee must be decidable at run time (1). Furthermore, wrappers must be applicable to any subtype of the static wrappee type (2).

An aggregate of a BorderWrapper wrapping a ButtonView should be insertable into a controls container, even though only the wrappee implements the interface IControl.

1. *Run-time applicability.* The actual type and instance of the wrappee must be decidable at run time.
2. *Genericity.* Wrappers must be applicable to any subtype of the static wrappee type.
3. *Transparency.* An aggregate should be an element of the wrapper and the actual wrappee type.
4. *Overriding.* Wrappers must be able to override methods of the wrappee.
5. *Shielding.* A wrapper should be able to control whether clients can directly access the wrappee.
6. *Safety.* The type system should prevent as many run-time errors as possible statically and signal errors as early as possible at run time.
7. *Modular reasoning.* Modular reasoning should be possible in the presence of wrapping.

Fig. 2. Requirements for a Wrapping Mechanism

Therefore, an aggregate should be an element of the actual wrappee type (3). This also implies that all methods of the wrappee can be called by clients and that they can make these calls directly on a reference to the wrapper.

Upon calling paint on an aggregate of a BorderWrapper and a TextView, the border's paint method should be executed. The latter first draws the border and then calls the paint method of the wrapped view with an adapted graphics context. Thus, wrappers must be able to override methods of the wrappee (4).

If clients can have direct references to the wrappee, they can call overridden methods. For example, a client could call the paint method of the embedded view with the graphics context (dimensions) of the whole aggregate. Hence, a wrapper should be able to control whether clients can directly access the wrappee (5).

Early detection of errors has already been identified as general requirements for component-oriented programming. We restate it here as an explicit requirement (6) for the purpose of assessing composition mechanisms.

The possibility for modular (component-wise) reasoning is another key requirement (7) for any mechanism targeted at component-based programming because of the independent development of components [30].

Finally, it is desirable that classes are not required to follow any coding standards for their instances to be wrappable. Otherwise, instances of classes programmed to different standards and of legacy classes are left out. Since certain coding standards can be established, as shown by JavaBeans, and since certain automatic rewriting —even of binary code— is possible, we consider this as a nice-to-have feature, but do not make it a formal requirement.

3 Why Existing Technology Is Insufficient

In this section we show why existing technology fails to address the above requirements.

Inheritance. Feature combination by multiple inheritance creates specialized combination classes, such as BorderedTextView and BorderedGraphicsView. Thus, it combines the functionality of the wrapper and the wrappee into a single object. However, combinations can only be made at compile time by a vendor having access to both the border

Technology \ Requirement	Run-time applicability (1)	Genericity (2)	Transparency (3)	Overriding (4)	Shielding (5)	Safety (6)	Modular reasoning (7)
Inheritance		(b)	✔	✔	n/a	✔	(c)
Parameterized mix-ins		(b)	✔	✔	n/a	✔	(c)
Containment	✔	(b)		(d)	(d)	(e)	✔
Specialized wrappers[(a)]	(f)	(b)	✔	✔	✔	✔	✔
Bottleneck interface	✔	✔		✔	✔		
Dual interface	✔	(b)		✔	✔	(e)	
Delegation in prototype-based languages	✔	✔	n/a	✔	✔		

(a) If only used with specific type, otherwise like containment.
(b) Yes, but with exceptions due to signature clashes.
(c) Limited due to tight coupling.
(d) Either full functionality availability or overriding and shielding.
(e) Type safety for static wrappee type.
(f) Type determined at compile time.

Fig. 3. Properties of Existing Technologies

and the view. Run-time feature composition, e.g., in compound documents, is impossible with inheritance. Hence, inheritance fails requirement (1). The modular reasoning requirement (7) is not fully satisfied because of the close coupling between super- and subclass, leading to the semantic fragile base class problem [19].

Mix-ins make it easier to create different combinations, even in single inheritance languages. However, combinations must be made at compile time. Thus, mix-ins also fail the requirement of run-time applicability (1).[1]

Containment. The containment approach, also known as the decorator pattern [9], has already been sketched along with the presentation of the example in Sect. 2.1. It's main problem is that the aggregate is not a subtype of the actual wrappee. Thus, it fails the transparency requirement (3).

Specialized wrappers can be created at compile time for specific known wrappee types. However, specialized wrappers aren't of much help in a component market with mutually unaware vendors.

Bottleneck and dual interfaces, i.e., a single message handler method, are variations of the containment approach. They make the full functionality of the actual wrappee available through the wrapper, but lose the benefits of the static type system. Thus, they fail both the safety (6) and the transparency (3) requirements.

Delegation in prototype-based languages. Prototype-based languages, such as Self [31], use a parent object to which the receiving object delegates messages that it does not understand itself. A bordered text view could be implemented by a border object with a text view parent. Due to the lack of (static) typing and because of the possibility to reassign the parent object at any time, prototype-based languages fail the requirements of safety (6) and modular reasoning (7) [9].

[1] Most genericity mechanisms, such as GJ, do not support mix-ins because they do not allow the type parameter to be used as a supertype of the parameterized type.

Summary. We conclude that none of the existing technologies gives a satisfactory solution to the problem at hand. Figure 3 summarizes the results. Less common solution approaches are described in Sect. 9. More detailed refutations appear in [4].

4 Generic Wrappers

To solve the problem stated in Sect. 2, we introduce generic wrappers. Generic wrappers are classes that are declared to wrap instances of a given reference type (class, interface) or of a subtype thereof. Like an extends clause to specify a superclass, we use a wraps clause to state the static wrappee type. This also declares the wrapper class to be a subtype of the static wrappee type. For example, the declaration

```
class LabelWrapper wraps IView {...}
```

states that each instance of the class LabelWrapper wraps an instance of a class that implements IView. The declaration makes class LabelWrapper a subtype of IView. Thus, instances of LabelWrapper can be assigned to variables of type IView and LabelWrapper has all public members (methods, fields) of IView.

To assure that this subtyping relationship always holds (and thereby that forwarding of calls never fails) must instances of LabelWrapper always wrap an instance of a subtype of IView —already during the execution of constructors. Hence, the wrappee must be passed as a special argument (in our syntax delimited by <>) to class instance creation expressions:

```
TextView t = ...; IView v = new LabelWrapper<t>(...);
```

The compiler checks that the declared type of variable t is a subtype of the static wrappee type. The wrapper class instance creation expression throws an exception if the value of t is null or if t were an expression and its evaluation throws an exception. In both cases, no wrapper object is created and the value of v remains unchanged.

The particularity of generic wrappers is that their instances are not only of the static, but also of the actual wrappee type. For example, a LabelWrapper wrapping a TextView is also of the latter type and not just of type IView. Hence, such an aggregate can be assigned to a variable of type TextView and the latter's methods can be called on it. In the following program fragment, which is based on the definition of LabelWrapper above, the type test returns true and the cast succeeds:

```
IView v = new LabelWrapper<new TextView()>(...);
TextView t2; if(v instanceof TextView) {t2=(TextView)v;}
```

Methods declared in the wrapper override those in the wrappee analogously to overriding in subclasses.

In constructors and instance methods of generic wrappers, the keyword wrappee references the wrappee. It can be treated like an implicitly declared and initialized final instance field. Hence, wrappers can call overridden methods of the wrappee using the keyword wrappee corresponding to super for overridden methods of superclasses. For example, the paint method of BorderWrapper might look as follows:

```
public void paint(Graphics g) {
  ...; // paint border
  wrappee.paint(g1); // paint wrapped view with adapted graphics context
}
```

Preliminary evaluation. Although this basic definition still leaves many aspect open, we can evaluate which requirements (Fig. 2) it fulfills independently of how the details are fixed. The actual type and instance of the wrappee can be decided upon at run time. Hence, requirement (1) is satisfied.

Wrappers are applicable to a all of the static wrappee type's subtypes, for which no unsound overriding would occur. Thus, the genericity requirement (2) is mostly fulfilled.

As defined above, instances of generic wrappers are members of the actual wrappee type. Therefore, the transparency requirement (3) is satisfied.

The fulfillment of the shielding (5) and modular reasoning (7) requirements cannot be judged without fixing more details.

The compiler ensures that an aggregate is always of the static wrappee type and, thereby, that all calls to methods of the static wrappee type will succeed. Run-time tests can be used to check whether the aggregate is of a certain type. Only insufficiently guarded casts may fail. Calls to methods of the actual wrappee type always find a matching method. Hence, the type system fulfills the safety requirement (6) by preventing as many run-time errors as possible statically and signaling errors as early as possible at run time.

5 Design Space for Generic Wrappers

The basic definition of generic wrappers in the previous section leaves many aspects open. In this section, we investigate the design space for generic wrappers.

The time of binding has a major influence on the design space of generic wrappers as compared to inheritance. With inheritance, the superclass is bound at compile time. With generic wrappers the actual type and instance of the wrappee first become known at wrap time, that is, run time. Later binding brings flexibility, but means that certain compatibility checks asserting type soundness and, thereby, the success of all method lookups have to be delayed (Fig. 4). A notable feature of generic wrappers is that an existing wrapper object can be wrapped again. Thus, it remains always possible to add new functionality to an aggregate.

Dynamic linking partly blurs this distinction. The name of the superclass is fixed at compile time, but the actual version and, therefore, the members and their semantics are not known until load time. For example in Java, the loading of a class may be delayed until an instance thereof is created. In this case, the compatibility with the used superclass is checked as late as the compatibility between a wrapper and the actual wrappee type. Thus, dynamic linking postpones compatibility checking to run time without fully exploiting the flexibility thereof.

5.1 Overriding of Instance Methods

Overriding of instance methods in subclasses is governed by certain rules to guarantee both type and semantic soundness. The same rules extend to overriding of methods

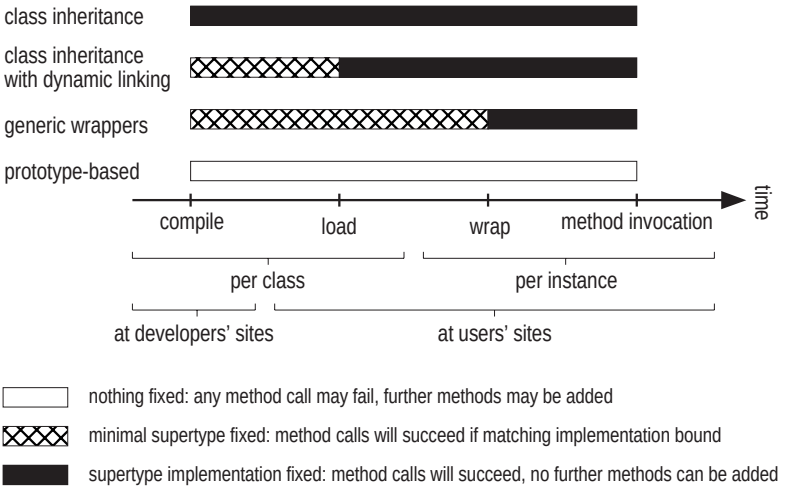


Fig. 4. What Is Asserted to Hold from Where on?

of the wrappee by methods of the wrapper. For example to guarantee type soundness in Java, the overridden method must not be final, the return type of the overriding method must be the same as that of the overridden method, the overriding method must be at least as accessible, the overriding method may not allow additional types of exceptions to be thrown, and an instance method may not override a class method. To also guarantee semantic soundness, the overriding method must be a behavioral refinement of the overridden method [1].

Although the actual type of the wrappee isn't known until wrap time, we can perform certain checks at compile time. We can check that overriding of methods of the static wrappee type by methods of the wrapper respect the above rules. Any violation of the type rules would necessarily also lead to a violation in combination with any actual wrappee type, i.e., a subtype of the static wrappee type.

Because the actual wrappee may have more methods than the static wrappee type, overriding conflicts may nevertheless occur at wrap time, i.e., when the combination of the wrapper and the wrappee first becomes visible. In Fig. 5, three overriding conflicts occur when wrapping an instance of A in an AWrapper. The methods A.m an A.o would be overridden by semantically incompatible ones and AWrapper.n cannot override A.n because they have different return types.

Below we discuss two approaches to this problem. The first checks type soundness at wrap time and throws an exception if wrapping would be type unsound. To decrease the probability of unsound overriding, we suggest a number of coding conventions. The second approach avoids wrap-time type problems by relying on a different form of method lookup and subsumption. We conclude with a short refutation of static approaches.

In this section we assume that there are no final classes and no method header specialization in subtypes (overriding non-final with final methods, overriding with restricted exception throws clauses and higher accessibility, as well as covariant return type and contravariant parameter type specialization) in our language. The interaction

```
interface IA {
  int m(); // return 0 or 1
}

class AWrapper wraps IA {
  public int m() {return 0;};
  public void n() {...};
  public int o() {return 0;};
}
```

```
class A implements IA {
  public int m() {return 1};
  public int n() {...};
  public int o() {return 1};
}

IA a=new A();
AWrapper w=new AWrapper<a>();
```

Fig. 5. Overriding Example

of final classes and method header specialization with generic wrappers is discussed in Sect. 6.2.

Wrap-time tests and coding conventions. At wrap time, we can automatically check whether overriding of methods of the actual wrappee by the wrapper is type sound. If this is the case, we can create the wrapper instance. Otherwise, we throw an exception. Wrap-time tests require enough information in the binary code. Java byte code, for example, satisfies this requirement.

Wrap-time exceptions are undesirable, yet they are preferable over unsuccessful method lookup as in prototype based languages like Self. First, if components are combined by an assembler, she can much more easily check all combinations than all method calls on all combinations. Second, if an error occurs, detecting it as early as possible facilitates debugging, as expressed by requirement (6).

To reduce the probability of wrap-time type conflicts, we could use laxer rules in analogy to Java's binary compatibility. However, laxer typing rules threaten semantic soundness, which must be the ultimate goal. Hence, we believe that the strict rules should be used for generic wrappers at wrap time also.

We suggest to adhere to the following four coding conventions, which can greatly reduce the possibility of both type and semantic conflicts at wrap time:

(a) Classes only define (non private) methods declared in implemented interfaces.
(b) No two interfaces, not related by extension, declare methods with the same signature.
(c) Interfaces have semantic specifications and methods in classes are semantic refinements of their correspondences in the implemented interfaces.
(d) Method calls are only made on variables of interface, but not class types.[2]

We analyze the conventions for method o of Fig. 5. Convention (a) implies that both AWrapper and A implement interfaces declaring a method o. Furthermore, (b) dictates that this must be the same interface, say IO. The idea of behavioral subtyping [1] is that interfaces have semantic specifications and that methods in subtypes are behavioral refinements of the corresponding methods in their supertype. Assuming that

[2] Self calls, which are of course also allowed, are discussed in Sect. 5.3.

both AWrapper.o and A.o are refinements of IO.o, we can deduce that both 0 and 1 are correct return values. Finally, condition (d) implies that a call x.o() may only be written for x of static type IO. In this case, the value 0 returned by the overriding method AWrapper.m meets our expectations.

If (a) or (b) is not adhered to, then a type conflict may occur as illustrated by method n of Fig. 5. If (c) is not adhered to, it could be that IO.o specifies the return value to be 1, which would not hold in the above case. Finally, if (d) is not respected, we could make a call x.o on a variable of type A. If x contained a reference to an AWrapper wrapping an A, we would get a return value of 0 although we expected 1.

Conventions (a) and (d) could easily be enforced by a programming language. Instead of (b) a language can require qualified notation for member access instead of merging namespaces of interfaces. Convention (c) requires semantic proofs and is, therefore, more difficult to check. These conventions also avoid semantic problems in the overriding in subclasses. Hence, they are implicitly advocated as good style for object-oriented programming [9,30].

In conclusion, wrap-time checking allows us to avoid type unsound overriding. Furthermore, adherence to some also otherwise beneficial coding conventions can greatly reduce the possibility of type or semantic unsound overriding.

An alternative form of method lookup. An alternative is to have the wrapper only override methods already present in the static wrappee type. In Fig. 5, this would mean that only A.m would be overridden by AWrapper.m.

Instead of overriding additional methods of the actual wrappee, in the example A.n and A.o, we allow an aggregate to contain multiple methods with the same signature and base the dispatch on run-time context information. In the simplest case, the dispatch is based on the static type of the receiver:

```
AWrapper w=new AWrapper<new A()>();
int i; i=w.o(); // executes AWrapper.o, i=0
A a=(A)w; i=a.o(); // executes A.o, i=1
```

In more general cases, the dispatch is not only based on static, but on run-time context information, i.e., an object's history of subsumptions. To illustrate this, assume that method o is declared in interface IO and that both AWrapper and A implement IO. In the following code fragment, added to the above, the static type of the receiver is in both cases IO, but different implementations are executed:

```
IO x;
x=w; i=x.o(); // executes AWrapper.o, i=0
x=a; i=x.o(); // executes A.o, i=1
```

The problem is that there are two occurrences of IO in the aggregate. Thus, we have to choose one for subsumption.[3] Multiple non-virtual inheritance in C++ has a similar semantics.

[3] In our case, we have already subsumed the aggregate to be of type AWrapper, respectively A. A true choice would be needed in the first line if w were of a subtype of both AWrapper and A, e.g., the compound type [AWrapper, A] [3].

In languages that do not support method header specialization (Sect. 6.2) or final classes, this form of method lookup avoids wrap-time exceptions. However, to also achieve semantic soundness, we still need to adhere to the above four coding conventions. Otherwise, we could execute a.m() in the fragment above and be surprised that we don't get 1 as result. The soundness problems caused by specialization could only be avoided by fully giving up overriding.

Method lookup and subsumption are more complex with this approach. Furthermore, adding this to a single-inheritance language with 'normal' method lookup and subsumption for inheritance, we end up with two different forms of method lookup and subsumption. The technicalities of this approach for the case of compile-time composition of mix-ins can be found in [8].

Refutation of static approaches. Here we briefly discuss why some approaches that avoid possible wrap-time conflicts at compile time and that are based on normal overriding have serious deficiencies.

Allowing the wrapper to only override methods of the static wrappee type, but not add additional methods would avoid the problem of unsound overriding of additional methods in the actual wrappee, e.g. A.n. However, not allowing additional methods in the wrapper would be a severe restriction, which would greatly reduce the usefulness of generic wrappers. Furthermore, this approach would fail in languages that support final classes or method header specialization (Sect. 6.2).

Negative type information [24] could express that subtypes of IA must not have a method, like n, that might be overridden in an unsound way. However, this would also mean that an aggregate of an AWrapper and a subtype of IA would not be of a subtype of IA. Furthermore, negative type information cannot be expressed in type systems of current languages.

Requiring the exact type of the wrappee to be known at compile time, a third approach, would contradict the requirement of run-time applicability (1).

5.2 Hiding of Fields and Class Methods

In many languages, fields and class methods are hidden rather than overridden in subtypes. Hiding of fields, if permitted, is not problematic because the hiding field may have a different type than the hidden field. The static wrappee type is used to access hidden fields in the actual wrappee. Hiding of class methods is usually governed by similar requirements as overriding of instance methods. Thus, the same two options apply.

5.3 Forwarding vs. Delegation

The difference between forwarding and delegation is the binding of the self parameter in the wrappee when called through the wrapper. With delegation, the self parameter is bound to the wrapper, with forwarding it is bound to the wrappee. Figure 6 illustrates the difference with a client calling method m of the wrappee on a reference to the wrapper.

Forwarding is a form of automatic message resending; delegation is a form of inheritance with binding of the parent (superclass) at run time, rather than at compile/link time as with 'normal' inheritance [16].

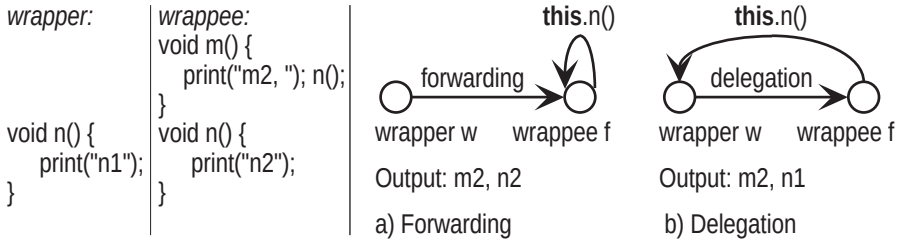


Fig. 6. Forwarding vs. Delegation

In all cases, super calls in the wrappee invoke methods of its superclass and not the wrapper's superclass. During these calls, this is bound to the wrappee with forwarding and to the wrapper with delegation.

Delegation has the advantage that the wrapper can better modify the behavior of the wrappee. With forwarding, the wrappee can better control its own behavior and avoid such problems as the semantic fragile base class problem. Generic wrapping as discussed in this paper works with both options. The reader is referred to [30,19,26] for further discussions of pros and cons.

5.4 Replacing a Wrappee

The wrappee of a given wrapper could be replaced by another object, the type of which is a subtype of the old dynamic wrappee type. It is not sufficient that the new wrappee is a subtype of the static wrappee type: A BorderWrapper wrapping a TextView can be referenced by a variable of static type TextView. Replacing the wrappee by a ButtonView would violate type soundness.

Although cyclic wrapping is type sound in combination with certain features, it is for semantic reasons mostly undesirable. Cyclic wrapping is prevented by the construction process, because the wrappee must be passed as an argument to the wrapper instance creation expression (Sect. 4). If we don't want cyclic wrapping, we also have to prevent it in the replacement of a wrappee.

For semantic reasons, we think that the wrappee should not be exchangeable. By fixing the wrappee for the lifespan of the wrapper, the system becomes more static and, therefore, simpler to analyze and reason about.

5.5 Direct Client References to the Wrappee

There are both advantages and disadvantages to allowing clients to hold direct references to the wrappee and being able to invoke the latter's methods —bypassing a possible overriding by the wrapper. On the positive side, this may give clients the possibility to invoke methods that are 'accidentally' overridden. For example, both BorderWrapper and TextView may define instance methods setColor with the same parameters. Without direct access to the wrappee, clients may not be able to change the text color. With direct access to the wrappee, this is possible. However, only clients that are aware that they reference a wrapped TextView rather than a bare one can do so. With the alternative

method lookup, TextView.setColor can also be accessed through a cast, unless the method is already declared in the static wrappee type IView.

The disadvantage of direct client references to the wrapper is that clients may invalidate invariants ranging over both the wrapper and the wrappee. Furthermore, we end up with different reference values to the same aggregate; thus, loosing the unique identity and the possibility of direct reference comparison.

The transparency of generic wrappers reduces the need for direct client references. In the containment approach (Sect. 3) all functionality that the dynamic wrappee type provides beyond the static wrappee type can only be made accessible by giving clients direct access to the wrappee. With generic wrappers, on the other hand, the full functionality of the dynamic wrappee type is available through the wrapper.

Whether we allow the wrapper to hand out direct references or not, we have the problems of existing references to the wrappee and of the wrappee handing out self references. Even if we in principle permit direct references, we may want to restrict them to clients that explicitly ask for them and are aware of the dangers.

Redirection of existing references. The problem of existing references vanishes if there aren't any. In analogy to aggregation in Microsoft's COM, we could require the wrappee to be created along with the wrapper and not to allow the wrappee's constructor to pass out self references. However, experience with COM showed that this approach is often too restrictive [22].

The second best case is a single reference to the object to be wrapped. In type systems with aliasing control [12] that can guarantee uniqueness of references we could restrict wrapping to unique references. This single existing reference to the wrappee, which is used as argument in the wrapper construction, could then either be redirected to the wrapper or be set to null. The restriction to unique references may severely limit the applicability of wrappers. Furthermore, aliasing control is not common.

For mainstream languages we see the following options:

1. Keep the references to the wrapped object unchanged. This is only an option if we allow direct references to the wrappee. Unfortunately, clients won't recognize if an object they refer to has been wrapped. Hence, they might unknowingly invoke overridden methods of the wrappee and, thereby, cause the aforementioned semantic problems.
2. Update all existing references to point to the wrapper. Thanks to the transparency of generic wrappers this is sound. Since the type of a reference can only be increased by wrapping, assumptions that a reference is at least of a certain type are not falsified.

Handing out of self references. Wrappees may pass out self references, e.g. for event listener registration. If we don't want direct client references to the wrappee or only allow the wrapper to hand them out, we need to address this issue. The draconian solution is to disallow the use of this in the wrappee except for member access. This is, however, very restrictive and excludes instances of legacy classes not adhering to this restriction.

Alternatively, we may define this in the wrappee to reference the wrapper except when used for member access.

5.6 Multiple Wrapping

There are two forms of multiple wrapping, *conjunctive* and *disjunctive* (Fig. 7). Conjunctive (also called additive or recursive) wrapping applies multiple wrappers around each other. For example, we might wrap a TextView in a ScrollWrapper and the latter with a BorderWrapper.

Disjunctive wrapping presents the same wrappee with different wrappers. It has analogous drawbacks as direct client references to the wrappee, namely the possibility of invalidating invariants ranging over the wrappee and one of its other wrappers. With type transparency, disjunctive wrapping can in most cases be replaced by conjunctive wrapping because the full dynamic wrappee type is visible through all wrappers.

If we allow direct client references to the wrappee but not disjunctive wrapping, we have to define what happens if a client wraps an object that is already wrapped. The options are disallowing it and throwing an exception if tried, putting the new wrapper between the wrappee and the old wrapper, and applying the new wrapper around the old wrappee. Thanks to the transparency of generic wrappers, all options are type sound.

5.7 Concealment

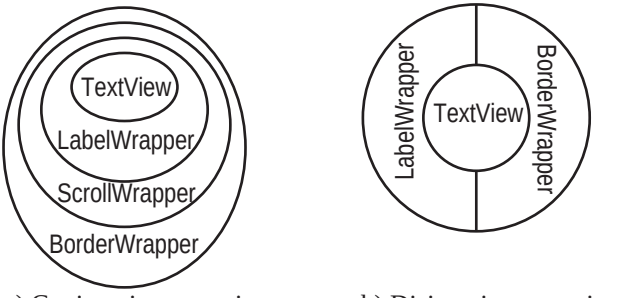
In certain cases, a wrapper may want to conceal part of the wrappee from clients. For example, a ConfidentialWrapper and its wrappee should not be serializable for confidentiality reasons. Thus, the wrapper wants to conceal the interface Serializable from clients in case the wrappee implements it. For this case, a conceals clause may be useful in combination with wraps:

```
class ConfidentialWrapper wraps IView conceals Serializable {...}
```

With this definition, no instance of a ConfidentialWrapper aggregate will ever be an element of Serializable.

Alternatively, a wrapper could be transparent for explicitly listed types only:

```
class SpecialWrapper wraps IView hoists IText, IGraphics {...}
```



a) Conjunctive wrapping b) Disjunctive wrapping

Fig. 7. Conjunctive and Disjunctive Wrapping

When such a SpecialWrapper wraps an instance of a class implementing IText then the functionality declared in IText can be accessed through the wrapper. On the other hand, if the same class also implements another interface, say IContainer, the latter's functionality cannot be accessed through the wrapper and the wrapper cannot be assigned to a variable of static type IContainer. Although transparency is restricted, this approach differs from the containment approach (Sect. 3) in that the type of the aggregate depends on the actual type of the wrappee.

Concealment may be practical for special cases, but it causes type soundness problems because the aggregate is not a subtype of the wrappee. Existing references to the wrappee cannot be redirected to the wrapper, if the latter conceals (part of) the static type of the variable containing the reference. Concealment also causes similar problems in combination with solutions 2 and 3 of applying a wrapper to an already wrapped object (Sect. 5.6). Furthermore, with delegation self calls of the wrappee to methods that are concealed by the wrapper fail. For this, a workaround would be to conceal types only from clients, but not from the aggregate itself.

These problems may, but do not necessarily occur in a given system that uses concealment. In analogy to Eiffel allowing subclasses to conceal[4] inherited members, we could allow concealment of types. This would, however, require system validity checks of complete systems.

5.8 Multiple Wrappees

So far, we have assumed that a given wrapper instance wraps exactly one object. This could be generalized to a fixed or arbitrary number of objects, thereby providing a single view of a subsystem implemented by multiple objects corresponding to the facade pattern [9]. Similar to multiple code inheritance, this works well unless different wrappees implement methods with the same signature and the wrapper does not override them. In this case, message lookup needs to be redefined. With run-time wrapping, such conflicts caused by type transparency may not be visible at compile time.

6 Interaction with Other Typing Mechanisms

In this section we discuss the interaction of generic wrappers with other common typing mechanisms.

6.1 Subclassing

Here we investigate whether and how generic wrappers can substitute inheritance and how the two may be combined.

[4] This is called 'hiding' in Eiffel. We don't use this term here to avoid confusion with Java style hiding of class methods and fields (Sect. 5.2).

Generic wrappers as a substitute for inheritance. If we choose delegation (Sect. 5.3) for generic wrappers, then they can be used to simulate class-based inheritance as follows:

```
class D extends C {...};
D d=new D();
```

Inheritance

```
class D wraps C {...};
D d=new D<new C()>();
```

Simulation with generic wrappers

The main difference is that at compile time we only know the lower bound of the wrappee type for generic wrappers, whereas with inheritance we know the exact superclass. This can be interpreted as flexibility or as lack of knowledge.

If, on the other hand, we use forwarding instead of delegation for generic wrappers, then we cannot modify the semantics of self calls in methods of the supertype. Thus, such generic wrappers cannot be used to simulate inheritance.

Subclassing of wrapper classes. In most mainstream languages subclassing implies subtyping. For this principle to extend to wrappers, a subclass of a wrapper class has to be declared to wrap the same type as its superclass or a subtype of its superclass' static wrappee type. Covariant specialization of the static wrappee type is possible unless the wrappee can be replaced using a method like setWrappee(StaticWrappeeType w), where the static wrappee type occurs in a contravariant position.

6.2 Method Header Specialization and Final Classes

Some languages allow overriding methods to have more specialized headers. For example, Java allows a non-final method to be overridden by a final one and allows the overriding method to have a more restricted exception throws clause and a higher accessibility. Other languages also allow covariant return type and contravariant parameter type specialization.

This creates problems with overriding by wrappers, even if the overriding is statically visible. Wrapping an instance of B with a BWrapper in Fig. 8, would override the final method B.p with an empty throws clause by BWrapper.p, which may throw SomeException.

To prevent such unsound overriding, we have to use wrap-time exceptions. Method header specialization is the type correspondent of semantic refinement discussed in Sect. 5.1.

Final classes pose a similar problem. They should not be subtyped. Thus, it is a compile-time error to declare a wrapper with a static wrappee type that is a final class type. At run time, an exception is thrown if an attempt is made to wrap an instance of a final class.

6.3 Parametric Types

Generic wrappers and parametric types can be combined without problems. Instances of generically derived classes don't distinguish themselves from instances of normal classes. Hence, they can be normally wrapped. Generic wrappers can also be used as bounds in generic classes and as actual parameters in generic derivations.

```
interface IB {
  void p() throws SomeException;
}

class BWrapper implements IB wraps IB {
  public void p() throws SomeException {...};
}

class B implements IB {
  public final void p() {...};
}

IB b=new B();
BWrapper w=new BWrapper<b>(); // illegal wrapping caught by exception
((B)w).p() // final method would be overridden and exception might be thrown
```

Fig. 8. Method Header Specialization Example

7 Generic Wrappers in Java

As a proof of concept, we add generic wrapping to Java. We present generic wrappers as a strict extension, that is existing Java programs need not be changed and instances of existing classes can be wrapped.

We select a consistent set of features from the aforementioned design choices and give a definition of generic wrappers in Java. We base our choices on the motivating examples and the above discussions, without repeating them. Next, we discuss selected integration issues with the Java library. Finally, we show how the defined mechanism solves the motivating problem. A discussion of efficient implementation strategies is beyond the scope and page limit of this paper.

7.1 Feature Selection and Language Integration

Both compile-time and wrap-time overriding and hiding are governed by the same rules as (compile-time) overriding and hiding in subclasses. Furthermore, we don't allow instances of final classes to be wrapped. Violations of these rules by the wrapper/static wrappee pair are flagged at compile time; violations by the wrapper/actual wrappee pair cause exceptions at the time of wrapping.

To get loose coupling between the wrapper and the wrappee and to facilitate semantic reasoning we chose forwarding over delegation and fix the wrappee for the lifespan of the wrapper. All existing references to the wrappee are redirected to the wrapper upon wrapping. We define this in instance method of the wrappee to refer to the (outermost) wrapper except when used for member access.

In a tribute to flexibility, we allow clients to explicitly attain direct references to the wrappee. The implementor of the wrapper class determines whether clients can get direct references to the wrappee by putting an access modifier (private, protected, public) between the keyword wraps and the static wrappee type, e.g:

```
class LabelWrapper3 wraps public IView {...}
```

The access modifier of the wrappee in a subclass must provide at least as much access as that in the superclass. The keyword wrappee can be treated like the name of a final instance field of the wrapper class with the used modifier, e.g public in the above example. To navigate back from a wrappee to its outermost wrapper, the method:

```
public final Object getWrapper() {return this;}
```

is added to the class Object. With the above definitions, this method returns a reference to the wrapper if the receiver object is wrapped and otherwise to the receiver itself.

We allow only conjunctive, but not disjunctive wrapping. Wrapping an already wrapped object corresponds to wrapping its outermost wrapper. Because it is not sound in combination with the above features, we don't allow concealment. Every wrapper has exactly one wrappee.

7.2 Library Integration

The library being an integral part of Java —the description of three packages is even part of the Java language specification— we discuss how serialization and cloning interplay with generic wrappers. Generic wrappers integrate in a straightforward way with most libraries, often providing new possibilities.

For instances of a Java class to be serializable, the class must implement the empty interface Serializable. A problem arises if only the wrapper or only the wrappee implement Serializable. In this case, the aggregate appears to be serializable although it isn't. In practice, the best solution is to throw a NotSerializableException when trying to serialize such an aggregate. In [4], we discuss certain options to statically avoid part of the problem.

Similar problems occur with cloning if only the wrapper or only the wrappee implements Cloneable. Because we don't allow disjunctive wrapping, clone has to create deep copies. Throwing a CloneNotSupportedException is again the best solution.

7.3 Assessment

Our mechanism fulfills all requirements (Fig. 2) except for genericity (2). The latter fails in cases where overriding would not be sound. We consider this acceptable because exceptions are already thrown at the time of wrapping —and not at the time of member access— and because creation of new instances can also fail for other reasons with an exception in existing Java.

Clearly, the motivating problems (Sect. 2.1) can be solved with the presented generic wrappers for Java.

8 Type Soundness

In this section, we report on a mechanically verified formal proof of type soundness of Java extended with generic wrappers. Type soundness intuitively means that all values

produced during any program execution respect their static types. An immediate corollary of type soundness is that method calls always execute a suitable method, that is, there are no 'method not understood' errors at run time.

Our proof of type soundness for generic wrappers is based on the work of von Oheimb and Nipkow [32]. They have formalized a large subset of Java and mechanically proved type soundness with the theorem prover Isabelle/HOL [23].

For this paper, we have added generic wrappers to this formalization,[5] adapted the proofs, and ran them through Isabelle/HOL. Here, we present the widening relations applicable to generic wrappers. A full report of all the mechanical details is beyond the scope of this paper.

The Java language specification [10] introduces identity and irreflexive widening conversions separately. 'Widening' is Java's form of subtyping. Since identity conversions are possible in all conversion contexts permitting widening, the two are merged in the formalization. The expression $\Gamma \vdash S \preceq T$ says that in program environment Γ objects of type S can be transformed to type T by identity or widening conversion. In particular, expressions of type S can be assigned to variables of type T and expressions of type S can be passed as formal parameters of type T.

We use the following naming conventions:

C, D	classes	A	list of classes
I, J	interfaces	S, T	arbitrary types
R	reference type	Γ	program, environment

The judgment $\Gamma \vdash C \prec_{\mathsf{C}} D$ expresses that class C is a subclass of class D, $\Gamma \vdash C \rightsquigarrow I$ that class C implements interface I, and $\Gamma \vdash I \prec_{\mathsf{I}} J$ that I is a subinterface of J. Furthermore, $\mathsf{is\_type}\ \Gamma\ T$ expresses that T is a legal type in Γ, $\mathsf{RefT}\ R$ denotes reference type R, and NT stands for the null type.

$\mathsf{Class}\ C$ stands for the class type C and $\mathsf{Iface}\ I$ for the interface type I. In our formalization we now have two kinds of classes: normal (non-wrapper) classes and wrapper classes. The discriminator $\mathsf{is\_wrapper}\ \Gamma\ C$ is true if C is a wrapper class. $\mathsf{WrappeeOf}\ \Gamma\ C$ denotes the the static wrappee type of class C in program Γ.

At run time, instances of wrapper classes are of aggregate types. Aggregate types are finite lists of at least two class types. An instance of the wrapper class C wrapping an instance of the wrapper class D that itself wraps an instance of the (non-wrapper) class E belongs to type $\mathsf{Aggregate}\ [C, D, E]$. The discriminator $\mathsf{is\_aggregate}\ \Gamma\ A$ is true if A denotes a possible combination of classes for an aggregate. Since there are no variables of aggregate type and because we do not allow the dynamic reassignment of wrappees, we only need widening rules with aggregates on the left-hand side of the conclusion judgment.

Furthermore, the discriminators $\mathsf{is\_class}\ \Gamma\ C$ and $\mathsf{is\_iface}\ \Gamma\ I$ are used. The following six typing judgments apply unchanged also to wrapper classes:

$$\frac{\mathsf{is\_type}\ \Gamma\ T}{\Gamma \vdash T \preceq T} \qquad \frac{\mathsf{is\_type}\ \Gamma\ (\mathsf{RefT}\ R)}{\Gamma \vdash \mathsf{NT} \preceq \mathsf{RefT}\ R}$$

[5] At http://www.abo.fi/~mbuechi/publications/GenericWrappers.html the Isabelle theories are available.

$$\frac{\Gamma \vdash I \prec_{\mathsf{i}} J}{\Gamma \vdash \mathsf{Iface}\ I \preceq \mathsf{Iface}\ J} \qquad \frac{\mathsf{is\_iface}\ \Gamma\ I; \mathsf{is\_class}\ \Gamma\ \mathsf{Object}}{\Gamma \vdash \mathsf{Iface}\ I \preceq \mathsf{Class}\ \mathsf{Object}}$$

$$\frac{\Gamma \vdash C \prec_{\mathsf{c}} D}{\Gamma \vdash \mathsf{Class}\ C \preceq \mathsf{Class}\ D} \qquad \frac{\Gamma \vdash C \rightsquigarrow J}{\Gamma \vdash \mathsf{Class}\ C \preceq \mathsf{Iface}\ J}$$

The following widening rules involving wrapper classes are used at compile time:

$$\frac{\mathsf{is\_wrapper}\ \Gamma\ C;\ \Gamma \vdash \mathsf{WrappeeOf}\ \Gamma\ C \preceq \mathsf{Class}\ D}{\Gamma \vdash \mathsf{Class}\ C \preceq \mathsf{Class}\ D}$$

$$\frac{\mathsf{is\_wrapper}\ \Gamma\ C;\ \Gamma \vdash \mathsf{WrappeeOf}\ \Gamma\ C \preceq \mathsf{Iface}\ J}{\Gamma \vdash \mathsf{Class}\ C \preceq \mathsf{Iface}\ J}$$

The following widening rules involving aggregates are used at run time (set converts a list into a set):

$$\frac{\mathsf{is\_aggregate}\ \Gamma\ A;\ \exists C \in \mathsf{set}\ A.\Gamma \vdash \mathsf{Class}\ C \preceq \mathsf{Class}\ D}{\Gamma \vdash \mathsf{Aggregate}\ A \preceq \mathsf{Class}\ D}$$

$$\frac{\mathsf{is\_aggregate}\ \Gamma\ A;\ \exists C \in \mathsf{set}\ A.\Gamma \vdash C \rightsquigarrow J}{\Gamma \vdash \mathsf{Aggregate}\ A \preceq \mathsf{Iface}\ J}$$

The main advantages of a mechanized over a paper-and-pencil proof are additional confidence and better support for extensions. We would like to stress the second aspect. Not only did the formalization result in a soundness proof, but the proof tool also reminded us of what all needed to be defined about generic wrappers before the desired properties could be established. Most proof scripts worked without modifications. The fact that all theorems were reproved mechanically[6] for the extended language definition conveys more confidence than the typical adaptation of a paper-and-pencil proof with 'this-should-still-hold' handwaving.

9 Related Work

Section 3 already provides an overview of some related mechanisms. With the exception of delegation, where a final comparison with our mechanism is deemed interesting, these technologies are not discussed again here. Comparisons with less closely related language mechanisms as well as binary component standards can be found in [4].

Delegation in prototype-based languages. What do we gain with generic wrappers over delegation in prototype-based languages? First, the static wrappee type and calls to it can be statically type checked. Some prototype-based languages, such as Cecil [5], also have (optional) static type systems. However, these languages require the exact type or even the concrete instance of the parent object to be known at compile time. The same approach is taken by prototype-based object calculi, e.g. [7]. Thus, they fail the requirement of run-time applicability (1).

[6] At the time of writing, a few lemmata have not yet been mechanically proved.

Second, with generic wrappers the dynamic wrappee type can be checked with runtime type tests. Third, type casts are the only points of failure; method lookup always succeeds. This greatly simplifies debugging by indicating errors closer to where they occur. Fourth, generic wrappers are targeted at mainstream class-based languages.

For our exemplary generic wrappers in Java, we have chosen a set of distinguishing features that facilitate modular reasoning. First we have forwarding rather than delegation. Second the wrappee is assigned snappily differentiating it from reassignable parent fields. Third, we disallow disjunctive wrapping. The latter is no problem because we get sharing of behavior from classes whereas prototype-based languages have to use shared parents for this.

Lava. Kniesel [15] has implemented an extension of Java with wrappers. The main difference to our generic wrappers is that in his proposal the aggregate is not a subtype of the actual, but only of the static wrappee type. Thus his proposal fails the transparency requirement (3) and is more limited in its applicability. Lava's wrappers are a form of the decorator pattern with automatically generated forwarding stubs and multiple wrappees combined with delegation. Wrappees can be reassigned, thereby, complicating semantic reasoning. The proposal is not type sound because the wrappees are assigned within the constructor. Independent extensibility, the focus of our proposal, is not well supported.

Delegation for software and subject composition. Harrison et al. [11] discuss options for different bindings of this in the decorator and facade patterns. They show how to implement delegation using either stored or passed pointers in class-based languages. Furthermore, they propose a declarative approach, to be used by component assemblers, permitting the binding of this to be customized on a per-method base. Their solution does not address the shortcomings of the decorator pattern with respect to our requirements. Namely, it does not provide for transparency (3).

Dynamic object specialization and reflective mix-ins. gbeta [6], a generalized version of BETA, supports two forms of dynamic inheritance through multiple inheritance. Dynamic object specialization is a dynamic modification of the structure of an existing object, preserving object identity. For example, the statement somePtn##->anObject## enhances the structure of anObject with the pattern somePtn. Furthermore, gbeta allows non-constant virtual types as superpatterns.

Because gbeta uses submethoding with INNER rather than overriding, it is not obvious how the mechanisms of gbeta could be transferred to more 'standard' object-oriented languages.

Mezini [18] presents a sophisticated, but complex approach to object evolution without name collisions. However, her work is untyped. Steyaert et al. [27] propose dynamic inheritance through mix-ins. The catch is that each object must contain a specification of all its potential enhancements. This renders their proposal inapplicable for mutually unaware component vendors.

A proposal for mix-ins, which allow types to be derived at run time, is presented in [4]. It is shown that —ignoring the use of the type parameter in places other than the extends clause— they correspond to a special kind of generic wrappers where the wrappee must be created along with the wrapper.

Objective C. Categories in Objective C [20] allow classes to be extended with a new set of methods/protocols independently of the original class definition. This compile-time mechanism corresponds to creating a subclass and globally replacing all occurrences of the superclass by the subclass. Categories modify whole classes, rather than individual objects. Categories do not fulfill the requirements of run-time applicability (1) and genericity (2).

Binary Component Adaption [13] provides for similar adaption of Java binaries as categories for Objective-C binaries. With respect to the problem at hand, it has the same shortcomings.

Aspect-oriented programming. Aspects [14] are a new category of programming construct that 'cross-cut' the modularity of traditional programming constructs. So an aspect can localize, in one place, code that deeply affects the implementation of multiple classes or methods. Aspects modify classes at compile time. Hence, they do not address the problems of run-time composition of objects created by different components from different vendors.

Mix-in calculus. Bono et al. have developed a formal calculus of classes and mix-ins [2]. Method declarations in mix-ins are explicitly marked as overriding an existing method or introducing a new method. The lower type bound (static wrappee type) is computed from the signature of a mix-in. Redefined methods give positive type information and new methods negative type information. Subtyping is determined by the types' structures. Negative type information is used to avoid mix-in-application-time exceptions.

10 Conclusions

Late composition of software components from different vendors is the essence of component software, enabling component markets and flexible reuse. One form of late composition is the combination of features implemented by different vendors into object-aggregates that appear as single objects to their clients. Our analysis shows, that existing technologies fail to fully unlock this power.

To remedy the problem, we have proposed generic wrappers, a typed form of dynamic inheritance. We have analyzed the design space with respect to both type soundness and semantic intuition, desirability, and consistency with existing mechanisms, such as subclassing. One options is forwarding instead of delegation to loosen the coupling and, thereby, avoid the semantic fragile base class problem. Another options is the snappy assignment of the wrappee to facilitate modular semantic reasoning.

As a proof of concept, we have chosen a consistent set of desirable features for a concrete mechanism, which we added to Java. Finally, we have given a mechanized proof of type soundness for the extended language. Additionally, the formalization provides an operational semantics for Java extended with generic wrappers.

Acknowledgments. David von Oheimb and Tobias Nipkow provided us with their formalization of Java and helped us with our extensions. We would like to thank Ralph Back, Dominik Gruntz, and Cuno Pfister for a number of fruitful discussions. The referees' helpful comments are also gratefully acknowledged.

References

1. Pierre America. Designing an object-oriented programming language with behavioral subtyping. In *Foundations of Object-Oriented Languages, REX School/Workshop*, pages 60–90. LNCS 489, Springer Verlag, 1991.
2. Viviana Bono, Amit Patel, and Vitaly Shmatikov. A core calculus of classes and mixins. In *Proceedings of ECOOP '99*, pages 43–66. LNCS 1628, Springer Verlag, 1999.
3. Martin Büchi and Wolfgang Weck. Compound types for Java. In *Proceedings of OOPSLA '98*, pages 362–373. ACM Press, 1998. http://www.abo.fi/~mbuechi/.
4. Martin Büchi and Wolfgang Weck. Generic wrapping. Technical Report 317, Turku Centre for Computer Science, March 2000. http://www.abo.fi/~mbuechi/.
5. Craig Chambers. The Cecil language: Specification & rationale (version 2.1). Technical report, University of Washington, March 1997.
6. Erik Ernst. *gbeta – a Language with Virtual Attributes, Block Structure, and Propagating, Dynamic Inheritance*. PhD thesis, Department of Computer Science, University of Aarhus, Denmark, 1999.
7. Kathleen Fisher and John C. Mitchell. Notes on typed object-oriented programming. In *Proceeding of Theoretical Aspects of Computer Software*, pages 844–885. LNCS 789, Springer Verlag, 1994.
8. Matthew Flatt, Shriram Krishnamurthi, and Matthias Felleisen. Classes and mixins. In *Proc. 25th ACM Symp. Principles of Programming Languages*, pages 171–183. ACM Press, 1998.
9. Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.
10. James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification*. Addison Wesley, 1996.
11. William Harrison, Harold Ossher, and Peri Tarr. Using delegation for software and subject composition. Technical Report RC-20946 (92722), IBM Research Division, T.J. Watson Research Center, August 1997.
12. John Hogg. Islands: Aliasing protection in object-oriented languages. In *Proceedings of OOPSLA '91*, pages 271–285. ACM Press, 1991.
13. Ralph Keller and Urs Hölzle. Binary component adaptation. In *Proceedings of ECOOP '98*, pages 307–329. LNCS 1445, Springer Verlag, 1998.
14. Gregor Kiczales et al. Aspect-oriented programming. In *Proceedings of ECOOP '97*, pages 220–242. LNCS 1241, Springer Verlag, 1997.
15. Günter Kniesel. Type-safe delegation for run-time component adaptation. In *Proceedings of ECOOP '99*. LNCS 1628, Springer Verlag, 1999.
16. Henry Lieberman. Using prototypical objects to implement shared behavior in object-oriented systems. In *Proceedings of OOPSLA '86*, pages 214–223. ACM Press, 1986.
17. Bertrand Meyer. *Eiffel: The Language*. Prentice Hall, second edition, 1992.
18. Mira Mezini. Dynamic object evolution without name collisions. In *Proceedings of ECOOP '97*, pages 190–219. LNCS 1241, Springer Verlag, 1997.
19. Leonid Mikhajlov and Emil Sekerinski. A study of the fragile base class problem. In *Proceedings of ECOOP '98*, pages 355–374. LNCS 1445, Springer Verlag, 1998.
20. NeXT Software, Inc. *Object-Oriented Programming and the Objective-C Language*. Addison-Wesley, 1993.
21. Object Management Group. CORBA components, 1999. Revision February 15, 1999, formal document orbos/99-02-01, http://www.omg.org.
22. Geoff Outhred and John Potter. Extending COM's aggregation model. In *Component-Oriented Software Engineering Workshop (in conjunction with the Australian Software Engineering Conference)*, 1998.

23. Lawrence C. Paulson. *Isabelle: A Generic Theorem Prover*. LNCS 828, Springer Verlag, 1994.
24. Didier Rémy. Typechecking records and variants in a natural extension of ML. In *Proc. 16th ACM Symp. Principles of Programming Languages*, pages 242–249. ACM Press, 1989.
25. Dale Rogerson. *Inside COM*. Microsoft Press, 1996.
26. A. Snyder. Encapsulation and inheritance in object-oriented programming languages. In *Proceedings of OOPSLA '86*, pages 38–45. ACM Press, 1986.
27. Patrick Steyaert and Wolfgang De Meuter. A marriage of class- and object-based inheritance without unwanted children. In *Proceedings of ECOOP '95*, pages 127–144. LNCS 952, Springer Verlag, 1995.
28. Bjarne Stroustrup. *The C++ Programming Language*. Addison Wesley, third edition, 1997.
29. Sun Microsystems, Inc. Java Beans, 1997. http://java.sun.com/beans/.
30. Clemens A. Szyperski. *Component Software – Beyond Object-Oriented Programming*. Addison-Wesley, 1997.
31. D. Ungar and R.B. Smith. Self: The power of simplicity. In *Proceedings of OOPSLA '87*, pages 227–241. ACM Press, 1987.
32. David von Oheimb and Tobias Nipkow. Machine-checking the Java specification: Proving type-safety. In Jim Alves-Foss, editor, *Formal Syntax and Semantics of Java*, pages 119–156. LNCS 1523, Springer Verlag, 1999.