Verifying Traits: A Proof System for Fine-Grained Reuse[*]

Ferruccio Damiani
Dipartimento di Informatica,
Università di Torino, Italy
damiani@di.unito.it

Johan Dovland,
Einar Broch Johnsen
Department of Informatics,
University of Oslo, Norway
{johand,einarj}@ifi.uio.no

Ina Schaefer
Technische Universität
Braunschweig, Germany
i.schaefer@tu-bs.de

ABSTRACT

Traits have been proposed as a more flexible mechanism for code structuring in object-oriented programming than class inheritance, for achieving fine-grained code reuse. A trait originally developed for one purpose can be modified and reused in a completely different context. Formalizations of traits have been extensively studied, and implementations of traits have started to appear in programming languages. However, work on formally establishing properties of trait-based programs has so far mostly concentrated on type systems. This paper proposes the first deductive proof system for a trait-based object-oriented language. If a specification for a trait can be given a priori, covering all actual usage of that trait, our proof system is modular as each trait is analyzed only once. In order to reflect the flexible reuse potential of traits, our proof system additionally allows new specifications to be added to a trait in an *incremental* way which does not violate established proofs. We formalize and show the soundness of the proof system.

Categories and Subject Descriptors

D.1.5 [**Programming Techniques**]: Object-oriented Programming; D.3.3 [**Programming Languages**]: Language Constructs and Features; F.3.3 [**Studies of Program Constructs**]: Type Structure

General Terms

Design, Languages, Theory

Keywords

Proof System, Incremental Reasoning, Program Verification, Trait

[*]The authors of this paper are listed in alphabetical order. This work has been partially supported by the Deutsche Forschungsgemeinschaft (DFG), the Italian MIUR (PRIN 2009 DISCO), the German-Italian University Centre (Vigoni program) and the EU project FP7-231620 HATS (http://www.hats-project.eu).

1. INTRODUCTION

With class inheritance, classes have two competing roles as generator of instances and as unit of reuse; in contrast, *traits* are pure units for fine-grained reuse [15]. Traits can be composed in an arbitrary order and the composite unit (class or trait) has complete control over conflicts that may arise and must solve them explicitly.

A trait is a set of methods, completely independent from any class hierarchy. Thus, the common methods of a set of classes can be factored into a trait. A trait originally developed for a particular purpose may be adapted and reused in a completely different context. This can lead to potentially undesired or conflicting program behavior. Various formulations of traits have been studied for JAVA-like languages (e.g., [9, 20, 23, 29, 31]). The recent programming language FORTRESS [1] (which has no class-based inheritance) has a trait construct, while the 'trait' construct of SCALA [25] is indeed a form of mixin. Research on ensuring properties of trait-based programs has so far mostly considered type systems (e.g., [9, 20, 28, 29, 31]). These approaches ensure that the composed program is type correct; i.e., all required fields and methods are present with the appropriate types.

This paper presents a compositional, deductive proof system for a trait-based JAVA-like language [8]. This proof system can be used to guarantee that programs obtained through the flexible adaptation and composition of traits satisfy critical requirements, by reasoning modularly and incrementally about traits, adaptation, and composition. As far as we know, no deductive proof system for trait-based languages has been proposed so far.

The challenge in developing a deductive proof system for traits is to support the *flexibility* offered by traits while providing an *incremental* and *compositional* reasoning system. Ideally, when traits are composed in a class, the trait specifications already provide enough information to ensure the contracts of the interfaces of that class. In this case, the actual usage of the trait corresponds to its originally intended usage as reflected in its original specification. This specification can be established by analyzing the trait only once in a modular way. However, the original trait specification may overly restrict the flexibility of trait reuse. In order to align the proof system with this flexibility, traits are associated with *sets of possible specifications*, and the applicable specification of a trait depends on its context of composition. New specifications may be added *incrementally* to a trait without violating previous specifications. When traits are composed in a class, the specification of the composed traits is selected from compatible specifications of its constituent traits. Hence, our proof system supports modular reasoning for traits when applicable, but extends this modularity to incremental reasoning when required for flexible trait reuse.

We develop an inference system for trait analysis which tracks specification sets for traits, when traits are modified and composed.

ID	::=	**interface** I **extends** $\overline{\texttt{I}}$ { $\overline{\texttt{S}}$; }	interface declaration
S	::=	I m ($\overline{\texttt{I}}$ $\overline{\texttt{x}}$)	method header
T	::=	Tb $\mid$ Tc	trait name
TD	::=	**trait** Tb **is** BTE $\mid$	basic trait declaration and
		trait Tc **is** CTE	composed trait declaration
BTE	::=	{ $\overline{\texttt{F}}$; $\overline{\texttt{S}}$; $\overline{\texttt{M}}$ }	basic trait expression
TAE	::=	Tb $\overline{\texttt{ao}}$	trait alteration expression
CTE	::=	TAE $\mid$ $\texttt{CTE}_1 + \texttt{CTE}_2$	composed trait expression
ao	::=	[**exclude** m] $\mid$ [m **aliasAs** m] $\mid$	trait alteration operation
		[f **renameTo** f] $\mid$ [m **renameTo** m]	
F	::=	I f	field
M	::=	S { **return** e; }	method
e	::=	x $\mid$ this.f $\mid$ e.m($\overline{\texttt{e}}$) $\mid$ **new** C($\overline{\texttt{e}}$) $\mid$ (I)e	expression
CD	::=	**class** C **implements** $\overline{\texttt{I}}$	class declaration
		by { $\overline{\texttt{F}}$; } **and** CTE	

Figure 1: The syntax of FRTJnf

This inference system adapts previous work on lazy behavioral subtyping [13], which developed an incremental inference system for late bound method calls by separating the required and provided behavior of methods, to trait modification and composition. The approach does not depend on a particular program logic. For simplicity, we use a Hoare-style notation to specify the pre- and postconditions of method definitions in terms of proof outlines and do not consider, e.g., class or trait invariants.

Section 2 presents a trait-based JAVA-like language and Section 3 a specification notation for traits. Section 4 introduces the proposed proof system for traits and discusses how it can be used to verify class and interface specifications. The formal inference system for program analysis is introduced in Section 5, which also shows soundness for the proof system. Related work is discussed in Section 6. Section 7 concludes the paper and discusses future work. Proof sketches of the main results are available in [11].

2. A TRAIT-BASED LANGUAGE

In the formulation of traits considered in this paper, a trait consists of *provided methods* (i.e., methods defined in the trait), *required methods* which parametrize the behavior, and *required fields* that can be directly accessed in the body of the methods. Traits are building blocks to compose classes and other, more complex, traits using a suite of trait composition and alteration operations. Since traits do not specify any state, a class assembled from traits has to provide the required fields of its constituent traits.

For the purpose of this paper, we use FRTJnf (FEATHERWEIGHT RECORD-TRAIT JAVA normal form), a calculus for traits within a JAVA-like nominal type system. The syntax of FRTJnf is given in Figure 1. A basic trait expression {$\overline{\texttt{F}}$; $\overline{\texttt{S}}$; $\overline{\texttt{M}}$} *provides* the methods $\overline{\texttt{M}}$ and declares the types of the *required* fields $\overline{\texttt{F}}$ and methods $\overline{\texttt{S}}$ (that can can be directly accessed by the bodies of the methods $\overline{\texttt{M}}$). The *symmetric sum* operation + merges two traits to form a new trait and requires that the summed traits are disjoint, i.e., they do not provide identically named methods (they may require a same field or method). Further, traits can be manipulated by the following trait alteration operations. The operation *exclude* forms a new trait by removing a method from an existing trait. The operation *aliasAs* forms a new trait by giving a new name to an existing method; in particular, when a recursive method is aliased, its recursive invocation refers to the original method. The operation *renameTo* creates a new trait by renaming all occurrences of a required field name or of a required/provided method name from an existing trait. A class is assembled from a trait expression by providing the required fields and a constructor. Classes further implement interfaces, specifying the methods that can be called on an instance of the class. For simplicity, we omit the class constructors from the syntax: each class is assumed to have a constructor of the form C($\overline{\texttt{J}}$ $\overline{\texttt{f}}$){this.$\overline{\texttt{f}} = \overline{\texttt{f}}$;}, where $\overline{\texttt{J}}$ $\overline{\texttt{f}}$ are all the fields of the class. In FRTJnf, the declaration of types, behavior, and the generation of instances are completely separated. Traits only play the role of units of behavior reuse and are not types. Class-based inheritance is not present, so classes only play the role of generators of instances. Interfaces are the only source language types.

In the examples, when needed, we will use standard imperative language features such as assignments this.f=e and x = e (for x different from this), conditionals, and while-loops.

EXAMPLE 2.1. *As an ongoing example, we consider a simple bank account implementation. The following trait* TAccount *provides the basic operations for inserting and withdrawing money:*

```
trait TAccount is {
  int bal; // req. field
  bool validate(int a); // req. mtd.
  void update(int y); // req. mtd.
  void deposit(int x) {this.update(x);}
  void withdraw(int id, int x){
    boolean v = this.validate(id);
    if (v) {this.update(-x);}}
}
```

A basic account CAccount *may then be defined as follows, where the additional trait* TAux *defines the auxiliary methods required by* TAccount*:*

```
interface IAccount {void deposit(int x);
                    void withdraw(int id, int x);}
trait TAux is {
  int bal; int owner; // req. fields
  void update(int y) {this.bal = this.bal + y}}
  boolean validate(int id) {return (id == owner);}
}
class CAccount implements IAccount
by {int bal; int owner;} and TAccount + TAux
```

Since traits define flexible units for reuse, different account behavior may be defined by combining TAccount *with different traits. For instance, the class* CFeeAccount *charges an additional fee whenever the balance is reduced:*

```
trait TFee is {
  int fee; int bal; // req. fields
  void basicUpd(int y); // req. mtd.
  void update(int y) { this.basicUpd(y);
    if (y<0) {this.bal = this.bal - this.fee}}
}
class CFeeAccount implements IAccount
by {int bal; int owner; int fee;} and
   TAccount + TFee + TAux[update renameTo basicUpd]
```

The public methods of an object are those listed in the interfaces implemented by its class; the other methods and fields are private to the object and can only be accessed through this. For instance, the only public members of classes CAccount and CFeeAccount are the methods deposit and withdraw.

The semantics of a class composed from traits is specified through the *flattening principle* [15, 23]. Flattening states that the semantics of a method introduced in a class through a trait should be identical to the semantics of the same method defined directly within a class. The flattening function $[\![\cdot]\!]$ (available in [11]) specifies the semantics of FRTJnf by translating a FRTJnf class declaration to a JAVA class declaration, and a trait expression to a sequence of method declarations.

FRTJnf is a subset of the prototypical language SWRTJ [8]. The SWRTJ type system supports the type-checking of traits in isolation from the classes or traits that use them, so that it is possible to type-check a method defined in a trait only once (instead of having to type-check it in every class using that trait). A distinguishing feature of SWRTJ w.r.t. the other formulations of traits within a JAVA-like nominal type system with this property [1, 20, 29, 31] is that SWRTJ fully supports method exclusion and method/field renaming operations (such as the formulation of traits by Reppy and Turon [28] in a structurally typed setting).

In particular, FRTJnf is a subset of FRTJ [6, 7], a minimal core calculus (in the spirit of FEATHERWEIGHT JAVA [17]) for SWRTJ. The FRTJnf subset of FRTJ represents a "normal form" that simplifies the analysis, since it ensures that trait summation happens as late as possible in a trait composition. In the sequel, we assume that programs are well-typed according to the FRTJ type system [6, 7].

3. SPECIFYING BASIC TRAITS

The proof system in this paper does not depend on a particular program logic. For simplicity in the presentation, we use Hoare triples $\{p\}t\{q\}$ [16], where p and q are assertions and t is a program statement. Triples $\{p\}t\{q\}$ have a standard partial correctness semantics [3, 4], adapted to the object-oriented setting; in particular, de Boer's technique using sequences in assertions addresses the issue of object creation [12]. If t is executed in a state where the precondition p holds and the execution terminates, then the postcondition q holds, after t has terminated. The derivation of triples can be done in any suitable program logic. Let PL be such a program logic and let $\vdash_{PL} \{p\}t\{q\}$ denote that $\{p\}t\{q\}$ is derivable in PL. We consider the following assertion language with assertions a defined by

$$a ::= \mathtt{\textbf{this}} \mid \mathtt{\textbf{return}} \mid \mathtt{\textbf{null}} \mid f \mid x \mid z \mid op(\overline{a}).$$

Here, **this** is the current object, **return** the current method's return value, f a program field, x a formal parameter, z a logical variable, and op an operation on data types. An *assertion pair* (p, q) is a pair of assertions such that p is a precondition and q a postcondition (for some sequence of program statements).

For a basic trait, the provided methods are annotated with assertion pairs, specifying desired guarantees. A guarantee of a provided method `m` may crucially depend on the behavior of the methods called by `m`. Consequently, we give a *proof outline* [26] for the body t of `m`, in which each method call is decorated with the behavioral requirement needed by `m` in order to fulfill the guarantee, e.g., $\{r\}\, n\, \{s\}$ for some called method `n`. Given such a proof outline, it is straightforward to verify that the guarantee holds for `m` under the assumption that all requirements can be met, by applying the rules of PL. Let $O \vdash_{PL} t : (p, q)$ denote that O is *a valid proof outline* for the guarantee (p, q), i.e., $\vdash_{PL} \{p\}\, O\, \{q\}$ holds when we assume that the requirements given in O are correct. Each *guarantee* of `m` has an associated set of *requirements* on other methods, derived from the proof outline. The guarantee of a method together with the associated set of requirements constitute a *method specification*.

A trait is designed for flexible reuse. For this reason, it may be difficult to specify the methods in the trait in a way which covers all possible future usage of the trait. There may be many possible guarantees for its provided methods, depending on the context of use. Different guarantees have different associated proof outlines, which give rise to different requirements on the called methods. Thus, a provided method can have several specifications reflecting different usage contexts. The initial specification reflects the original intended usage of the method; new specifications may be added if new ways of using the trait are found later. If the initial specification happens to suffice for later usage, this is the special case which *coincides with modular specification*. The *specification of a trait* consists of the method specifications of its provided methods.

EXAMPLE 3.1. *Consider the* `withdraw` *method of trait* `TAccount` *in Example 2.1. This method may be given the following two specifications, labelled* `w1` *and* `w2`*:*

```
void withdraw(int id, int x){
  boolean v = this.validate(id);
  if (v) {this.update(-x);}}
  // w1: $(bal == b_0 \wedge id == owner, bal == b_0 - x)$
  //  reqs.: $\{bal == b_0\}$update(y) $\{bal == b_0 + y\}$,
  //      $\{id == owner\}$validate(id) $\{$return$ == $true$\}$
  // w2: $(bal == b_0 \wedge id \neq owner, bal == b_0)$
  //  req.: $\{id \neq owner\}$validate(id) $\{$return$ == $false$\}$
```

where trivial specifications, e.g., that `bal` *is not modified by* `validate`*, are omitted for brevity. For trait* `TAux`*, we may supply the following specifications, leading to no requirements:*

```
trait TAux is { int bal; int owner; // req. fields
  void update(int y) {this.bal = this.bal + y}}
    // $(bal = b_0, bal = b_0 + y)$

  boolean validate(int id) {return (id == owner);}
    // (true, return $== (id == owner)$)
}
```

4. COMPOSITIONAL VERIFICATION

The goal of our verification technique is to reason incrementally about trait expressions while verifying trait-based programs. Due to the flexible reuse potential of traits, we do not assume that a fixed specification of a trait, given a priori, covers all potential usages of that trait, although this is a special case of our more general incremental approach. Instead, traits provide a set of possible method specifications for each provided method that can be incrementally extended. Thus, we devise compositional proof rules that apply to sets of method specifications when traits are composed.

During the verification of a trait expression, a *trait environment* is constructed to keep track of the specifications for the provided methods. We assume that every method in an interface is annotated with an *interface contract* that is an assertion pair describing the behavior guaranteed by all implementations of that method to reason about calls to methods on interface types. In the following, we examine the different cases for constructing the trait environment for basic traits and composite traits constructed using the trait composition and modification operations of FRTJnf.

Basic Traits. Let `m` be a method provided by a basic trait `T`, and let a guarantee of `m` be given by the assertion pair (p, q). The requirements that `m` imposes on the called methods, result in a proof outline O for `m` with guarantee (p, q). Using this proof outline, we establish the guarantee (p, q) for `m` by deductive techniques; e.g., using KeY [5] to verify $O \vdash_{PL} t : (p, q)$. Method calls may be either *external* or *internal*. External calls rely on the interface of the callee, so they may be analyzed directly using interface contracts. For the proof outline O, we collect the requirements $\{r_i\}\, n_i\, \{s_i\}$ for internal calls in O together with the guarantee (p, q) as a specification for `m` in the trait environment for `T`. A provided method `m` may have different guarantees depending on different requirements on its called methods. Each of these guarantees is proven using a different proof outline, leading to different specifications for `m`.

Symmetric Sum of Traits. For two traits composed by symmetric sum, we keep the distinction that each method specification has particular assumptions on the required methods such that the trait environment of the composed trait is the union of the trait environments of the single traits. In particular, method specifications are kept in the trait environment even if their requirements cannot be satisfied by the implementations found in other traits in the composition. The reason is that the composed trait may be the subject to later trait composition or modification operations. Thus, method specifications that were unsatisfiable in the original composition may again be satisfiable. However, if the composed trait is used in a class definition, the analysis of the class ignores method specifications that are unnecessary in order to verify the interface contract of the class, thereby selecting a set of consistent method specifications from the set of all provided method specifications from the constituents of the composed trait.

Trait Modifiers. *Excluding* a method from a trait does not generate any proof obligations. The trait environment of the resulting trait is obtained from the previous trait environment by removing the method specifications of the removed method. *Aliasing* does not generate proof obligations. The trait environment for the resulting trait is obtained by copying the method specifications of the aliased method. *Renaming of methods* does not generate proof obligations, but proof obligations for distinct methods may now apply to the same method. The trait environment for the resulting trait is obtained by consistently renaming the respective method. Also *renaming of fields* does not generate proof obligations. The trait environment for the trait resulting by the trait alteration [f **renameTo** f′] is obtained by distinguishing different cases. If f does not occur in the previous trait, the trait environment for the resulting trait is obtained directly from the previous trait environment. Otherwise, for each method m, we consider whether f′ occurs in the body of m or not. In the former case, the method specifications of m are simply dropped. In the latter case, the method specifications of m containing occurrences of f′ are dropped and, in the other method specifications of m, the occurrences of f are renamed to f′.

EXAMPLE 4.1. *Let the trait* TAux *as specified in Example 3.1, and consider the rename operation* TAux[update **renameTo** basicUpd]. *The implementation of* validate *is unaltered by the operation, but the* update *method is renamed, and the specification given in* TAux *applies to the new method:*

```
void basicUpd(int y) {this.bal = this.bal + y}}
  // $(bal = b_0, bal = b_0 + y)$
```

Trait-based Classes. The main goal of the verification process for trait-based programs is to show that *a class implements the contracts of its declared interfaces.* It is necessary to show that every public method exposed through an interface guarantees the contract in that interface. This result should eventually follow from the specification of the methods, as provided by the trait expression. In the case where the trait specifications contain sufficient guarantees, this follows directly. Otherwise, new method specifications with additional guarantees may be added to trait specifications at need, and method specifications are collected when a class is assembled from traits by trait composition. For each added method specification, the respective method must be reinspected with a new proof outline. Such proof outlines may lead to new requirements on internally called methods, which makes it necessary to supply new proof outlines for these methods. This procedure repeats for auxiliary internal calls until the analysis is complete. Remark that all proofs which rely on the previously established guarantees of the provided method remain valid. Thus, the presented approach is incremental.

By the successful analysis of a class, the interface contract of every public method is guaranteed by the trait specifications, and the requirements on internal calls are guaranteed by the specifications of the called methods.

EXAMPLE 4.2. *Consider the analysis of class* CAccount *in Example 2.1, which is implemented by* TAccount + TAux. *Assume that in order to implement the interface* IAccount, *the proof obligation:* $(bal = b_0, (id == owner \Rightarrow bal == b_0 - x) \wedge (id \neq owner \Rightarrow bal == b_0))$ *must be verified for method* withdraw *in* TAccount. *Since this obligation follows by entailment from the guarantees of* w1 *and* w2 *in Ex. 3.1, it suffices to ensure that the requirements of* w1 *and* w2 *are satisfied for the implementations found in* TAux, *which is straightforward.*

Consider next the analysis of class CFeeAccount, *implemented by* TAccount + TFee + TAux[update **renameTo** basicUpd], *and assume that the proof obligation* $(x > 0 \wedge bal = b_0, (id == owner \Rightarrow bal == b_0 - x - fee) \wedge (id \neq owner \Rightarrow bal == b_0))$ *is imposed on* withdraw *by the interface* IFeeAccount. *Remark that this proof obligation does not follow from the guarantees of* w1 *and* w2, *and a new proof outline must then be analyzed. It suffices to extend the specifications of* withdraw *with the following specification* w3*:*

```
void withdraw(int id, int x) {...}
// w3: $(x > 0 \wedge bal = b_0 \wedge id = owner, bal == b_0 - x - fee)$
//   reqs.: $\{bal == b_0 \wedge y < 0\}$update(y)$\{bal == b_0 + y - fee\}$,
//       $\{id == owner\}$validate(id)$\{$return $==$ true$\}$
```

Since the above proof obligation for withdraw *follows by entailment from the guarantees of* w2 *and* w3, *it suffices to verify the requirements of these specifications. The only non-trivial requirement is the one to* update, *which can be verified by the following specification in* TFee*:*

```
trait TFee is {...
  void update(int y) {...}
    // $(y < 0 \wedge bal = b_0, bal = b_0 + y - fee)$
    //    req.: $\{bal = b_0\}$basicUpd(y)$\{bal = b_0 + y\}$
}
```

This requirement follows from the guarantee of basicUpd *as given in Ex. 4.1.*

5. THE INFERENCE SYSTEM

This section presents *PST(PL)*, a *P*roof *S*ystem for *T*rait-based programs which is parametric in the underlying program logic *PL*. The calculus *PST(PL)* relies on a sound program logic *PL*, and is defined by the inference rules given in [11]. Judgements in the calculus are of the form $\mathscr{C}, \mathscr{E} \vdash \mathscr{P}$, where $\mathscr{E}$ is a *trait environment* for trait analysis, $\mathscr{C}$ is a *class environment* for keeping track of declarations and specifications while analyzing classes, and $\mathscr{P}$ is a sequence of *analysis operations*. Initially, the trait and class environments are empty, and $\mathscr{P}$ is a sequence of trait and class definitions. For each analyzed trait, the trait environment $\mathscr{E}$ is extended with the trait definition and the specifications for the defined methods. Thus, if the analysis of a trait T is initiated in some trait environment $\mathscr{E}$, the successful analysis of T will lead to some trait environment $\mathscr{E}'$ which is an extension of $\mathscr{E}$. In this case. we say that $\mathscr{E}'$ is the trait environment *resulting* from the analysis. When analyzing classes, the class environment is extended similarly. Traits and classes are analyzed based on the trait and class environments resulting from the analysis of previous traits and classes.

Trait Analysis. Method guarantees are written as assertion pairs (p, q) of type *Guar*. To satisfy its guarantee, a method m may impose *requirements* of type *Req* on the called methods $\mathtt{n}_i$, of the form $\{r\}\ \mathtt{n}_i\ \{s\}$. A *method specification* of type *Spec* is a tuple $\langle Guar, Set[Req] \rangle$; i.e., a specification associates a set of requirements with the guarantee. If $\langle (p,q), \overline{R} \rangle$ is a specification for some method m, we say that m guarantees (p, q) assuming that the requirements $\overline{R}$ are satisfied for the called methods. Method specifications may be decomposed by the functions $guar : Spec \rightarrow Guar$ and $req : Spec \rightarrow Set[Req]$ where $guar(\langle (p,q), \overline{R} \rangle) \triangleq (p,q)$ and $req(\langle (p,q), \overline{R} \rangle) \triangleq \overline{R}$. These functions are straightforwardly lifted to sets of method specifications, returning sets of guarantees and requirements, respectively. Given a proof outline O, the function $reqs(O)$ returns the set of requirements occurring in O. Trait environments $\mathscr{E}$ of type Env are defined as follows:

Definition 5.1 (Trait environments). *A trait environment $\mathscr{E} : Env$ consists of two mappings $T_{\mathscr{E}}$ and $S_{\mathscr{E}}$, where $T_{\mathscr{E}} : \mathtt{TAE} \rightarrow \mathtt{BTE}$ and $S_{\mathscr{E}} : \mathtt{TAE} \times Mid \rightarrow Set[Spec]$.*

Mapping $T_{\mathscr{E}}$ takes a trait alteration expression and returns a basic trait expression, and mapping $S_{\mathscr{E}}$ takes a trait alteration expression and a method name and returns a set of method specifications. For each basic trait of the form **trait** $\mathtt{Tb}$ **is** $\{\overline{\mathtt{F}}; \overline{\mathtt{S}}; \overline{\mathtt{M}}\}$, the $T_{\mathscr{E}}$ mapping is extended with the definition of the trait, and each user given guarantee leads to a specification recorded by the $S_{\mathscr{E}}$ mapping. If for instance a guarantee (p,q) is given for method $\mathtt{I}\ \mathtt{m}(\overline{\mathtt{I}}\ \overline{\mathtt{x}})\{t\}$, a proof outline O must be supplied such that $O \vdash_{PL} t : (p,q)$, and the specification $\langle (p,q), reqs(O) \rangle$ is included in the set $S_{\mathscr{E}}(\mathtt{Tb}, \mathtt{m})$. Remark that the actual implementation that an internal call will bind to is not known when the trait is defined, since method binding depends on how the trait is used to form classes. Consequently, the imposed requirements are not verified with regard to any implementation during trait analysis; Requirements are only verified at need when the specification is actually used during class analysis. When analyzing a composed trait of the form **trait** $\mathtt{Tc}$ **is** $\mathtt{TAE}_1 + \ldots + \mathtt{TAE}_n$, where each $\mathtt{TAE}_i$ is of the form $\mathtt{Tb}_i\overline{\mathtt{ao}}_i$, properties for each $\mathtt{TAE}_i$ is remembered by the trait environment. By the successful analysis of $\mathtt{Tc}$, the mapping $T_{\mathscr{E}}$ takes $\mathtt{TAE}_i$ to a basic trait definition containing the methods provided by $\mathtt{TAE}_i$, and $S_{\mathscr{E}}$ contains specifications for these methods. These entities are derived by manipulating the entities of $\mathtt{Tb}_i$ according to the modifiers $\overline{\mathtt{ao}}_i$ as described in [11].

For each specification $\langle (p,q), \overline{R} \rangle$ included in $S_{\mathscr{E}}(\mathtt{TAE}, \mathtt{m})$, the inference system for trait analysis ensures that $\mathtt{m}$ is provided by $\mathtt{TAE}$ and that there exists a verified proof outline O for the body t of $\mathtt{m}$ such that $O \vdash_{PL} t : (p,q)$ where $reqs(O) = \overline{R}$.

Class Analysis. In addition to the trait environment $\mathscr{E}$, class analysis builds a class environment reflecting the definitions and specifications of classes. Classes are represented by a unique name and a tuple $\langle \overline{\mathtt{I}}, \mathtt{CTE}, \overline{\mathtt{F}} \rangle$ of type $Class$.

Definition 5.2 (Class environments). *A class environment $\mathscr{C}$ consists of two mappings $D_{\mathscr{C}}$ and $S_{\mathscr{C}}$, where $D_{\mathscr{C}} : Cid \mapsto Class$, and $S_{\mathscr{C}} : Cid \times Mid \mapsto Set[Spec]$.*

Here, $D_{\mathscr{C}}$ reflects the definitions of verified classes, and $S_{\mathscr{C}}$ reflects their verified specifications. The main purpose of the class environment is to record the method specifications used to establish the contracts of the implemented interfaces; The analysis of class **class** $\mathtt{C}$ **implements** $\overline{\mathtt{I}}$ **by** $\{\ \overline{\mathtt{F}};\ \}$ **and** $\mathtt{TAE}_1 + \ldots + \mathtt{TAE}_n$ is driven by contracts of $\overline{\mathtt{I}}$. By type-safety, we have that each provided method $\mathtt{m}$ is defined in exactly one $\mathtt{TAE}_i$. Upon the successful analysis of $\mathtt{C}$, each interface contract for some provided method $\mathtt{m}$ follows by *entailment* from the guarantees of $S_{\mathscr{C}}(\mathtt{C}, \mathtt{m})$. If $\mathtt{m}$ is provided by $\mathtt{TAE}_i$, the interface contracts are ensured by reusing already verified specifications contained in $S_{\mathscr{E}}(\mathtt{TAE}_i, \mathtt{m})$, and possibly extending $S_{\mathscr{E}}(\mathtt{TAE}_i, \mathtt{m})$ with new specifications if needed. Thus, $S_{\mathscr{C}}(\mathtt{C}, \mathtt{m})$ contains the subset of $S_{\mathscr{E}}(\mathtt{TAE}_i, \mathtt{m})$ that is actually used to verify the current class. In addition, the requirements imposed by the used specifications are analyzed with regard to the implementation they bind to for $\mathtt{C}$. Thus, if $\langle (p,q), \overline{R} \rangle \in S_{\mathscr{C}}(\mathtt{C}, \mathtt{m})$, then each requirement $\{r\}\ \mathtt{n}\ \{s\} \in \overline{R}$ follows by entailment from the guarantees of $S_{\mathscr{C}}(\mathtt{C}, \mathtt{n})$. The definition of the entailment relation $\rightharpoonup$ can be found in [11, 13].

Soundness. When reasoning about a set of mutually recursive methods, the guarantees in the specifications of all methods are assumed to hold in order to verify the body of each methods (e.g., [4]). We now extend this approach to define the *consistency* of a set of proof outlines for methods in a flattened class with given interfaces. The flattened version of **class** $\mathtt{C}$ **implements** $\overline{\mathtt{I}}$ **by** $\{\ \overline{\mathtt{J}\ \mathtt{f}};\ \}$ **and** $\mathtt{CTE}$ is given by **class** $\mathtt{C}$ **implements** $\overline{\mathtt{I}}\ \{\ \overline{\mathtt{J}\ \mathtt{f}};\ \mathtt{C}(\overline{\mathtt{J}\ \mathtt{f}})\{\mathtt{this}.\overline{\mathtt{f}} = \overline{\mathtt{f}};\}\ \overline{\mathtt{M}}\ \}$ as defined in [11].

Definition 5.3 (Consistency). *Consider the flattened class* **class** $\mathtt{C}$ **implements** $\overline{\mathtt{I}}\ \{\ \overline{\mathtt{J}\ \mathtt{f}};\ \mathtt{C}(\overline{\mathtt{J}\ \mathtt{f}})\{\mathtt{this}.\overline{\mathtt{f}} = \overline{\mathtt{f}};\}\ \overline{\mathtt{M}}\ \}$. *For each method $\mathtt{m} \in \overline{\mathtt{M}}$ with method body t, let $S_{\mathtt{m}}$ be a set of method specifications such that for each $\langle (p,q), \overline{R} \rangle \in S_{\mathtt{m}}$, there exists a proof outline O where $O \vdash_{PL} t : (p,q)$ and $\overline{R} = reqs(O)$. The specifications $S_{\overline{\mathtt{M}}}$ are* consistent *iff, for all $\mathtt{m} \in \overline{\mathtt{M}}$:*

1. $\forall \{r\}\ \mathtt{m}\ \{s\} \in contracts(\overline{\mathtt{I}})\ .\ guar(S_{\mathtt{m}}) \rightharpoonup (r,s)$
2. $\forall \langle (p,q), \overline{R} \rangle \in S_{\mathtt{m}}\ .\ \forall \{r\}\ \mathtt{n}\ \{s\} \in \overline{R}\ .\ guar(S_{\mathtt{n}}) \rightharpoonup (r,s)$

Here, the first condition expresses that the interface contracts are satisfied, whereas the second condition expresses that the requirements of all internal calls are satisfied. Previous work [13] defines a sound calculus for analyzing single inheritance class hierarchies. Given a consistent set of specifications, the analysis of a flattened class succeeds in this calculus. In order to ensure soundness of *PST(PL)*, it thereby suffices to prove that the successful analysis of some class $\mathtt{C}$ leads to a consistent set of specifications for the flattened version of $\mathtt{C}$. The proof of this theorem can be found in [11].

Theorem 5.4. *For a given class* **class** $\mathtt{C}$ **implements** $\overline{\mathtt{I}}$ **by** $\{\ \overline{\mathtt{F}};\ \}$ **and** $\mathtt{CTE}$, *if the successful analysis of $\mathtt{C}$ in PST(PL) leads to a class environment $\mathscr{C}$, then the set of method specifications for $\mathtt{C}$ in $\mathscr{C}$ are consistent for the flattened version of $\mathtt{C}$.*

6. RELATED WORK

An important factor for the success of class-based object-oriented programming is the inheritance mechanism to structure and reuse code. *Single inheritance* is the best supported by formal systems for program analysis. In the context of single inheritance, behavioral reasoning about extensible class hierarchies with late bound method calls is often performed in the context of *behavioral subtyping* (see, e.g., [2, 19, 21, 27]). Behavioral subtyping is an incremental reasoning strategy in the sense that a subclass may be analyzed in the context of previously defined classes. In order to avoid reverification of superclasses, method overridings must preserve the specifications of overridden methods. This approach has also been used for SCALA's 'trait' construct, but "significantly reduce the applicability and thereby benefits of traits" [30]. *Lazy behavioral subtyping* [13] is an incremental reasoning strategy for more flexible code reuse than behavioral subtyping. With lazy behavioral subtyping, the *requirements* that a method guarantee imposes on late bound method calls are identified, and the main idea is that there is no need to preserve the full specifications of overridden methods. In order to avoid reverification of superclass methods, only the weaker *requirements* imposed on late bound method calls need to be preserved by method redefinitions in subclasses.

Multiple inheritance is widely used in modeling notations such as UML [10], to capture that a concept naturally inherits from several other concepts. Versions of multiple inheritance are found in C++, CLOS, Eiffel, and Ocaml. Creol [18] has proposed a so-called healthy binding strategy which resolves horizontal name conflicts by avoiding accidental overridings. The proof systems presented in [14, 22, 24, 32] are the only proof systems we know for multiple inheritance class hierarchies. The work in [24] presents a Hoare-style program logic for Eiffel that handles multiple inheritance based on an existing program logic for single-inheritance by extending the method lookup definition. In [22], method calls are assumed to be fully qualified in order to avoid ambiguities, and diamond inheritance is not considered. In [32], ambiguities are assumed to be resolved by the programmer, a method can only be inherited if it has the same implementation in all parents. In contrast, [14] applies lazy behavioral subtyping to multiple inheritance and shows that healthy method binding is sufficient to allow incremental reasoning about multiple inheritance class hierarchies. The work in [14] resembles that of the current paper in the separation of concerns between required and guaranteed assertions for method calls and definitions, respectively. The main challenges for reasoning about class hierarchies with multiple inheritance are related to late bound method calls. In contrast, traits do not support late binding but the flexible composition of traits necessitates a delayed selection of relevant method specifications. Technically, this makes the two approaches fairly different. We are not aware of any previous proposal for a deductive proof system for a trait-based language.

7. CONCLUSION AND FUTURE WORK

This paper proposes a deductive proof system for trait-based object-oriented programs, reflecting the fine-grained reuse potential of traits at the level of reasoning. The approach focusses on verifying interface contracts. We plan an extension to additionally consider invariants, both at the level of classes and traits. A trait invariant may, for instance, capture relations between the required fields of a trait which extends the range of properties that can be incrementally verified for trait-based programs. Further, we plan to extend the KeY system [5] for deductive verification of JAVA programs to SWRTJ programs and to implement the proof system proposed in this paper within KeY.

Acknowledgements. We are grateful to Wolfgang Ahrendt, Richard Bubel, Olaf Owe, and Volker Stolz for valuable discussions on the subject of this work. We also thanks the anonymous FTfJP referees for insightful comments and suggestions.

8. REFERENCES

[1] E. Allen, D. Chase, J. Hallett, V. Luchangco, G.-W.Maessen, S. Ryu, G. Steele, and S. Tobin-Hochstad. The Fortress Language Specification, V. 1.0, 2008.

[2] P. America. Designing an object-oriented programming language with behavioural subtyping. In J. W. de Bakker, W.-P. de Roever, and G. Rozenberg, editors, *Foundations of Object-Oriented Languages*, pages 60–90. Springer, 1991.

[3] K. R. Apt. Ten years of Hoare's logic: A survey — Part I. *ACM Transactions on Programming Languages and Systems*, 3(4):431–483, Oct. 1981.

[4] K. R. Apt, F. S. de Boer, and E.-R. Olderog. *Verification of Sequential and Concurrent Systems*. Texts and Monographs in Computer Science. Springer, 3rd edition, 2009.

[5] B. Beckert, R. Hähnle, and P. H. Schmitt, editors. *Verification of Object-Oriented Software: The KeY Approach*, volume 4334 of *LNCS*. Springer, 2007.

[6] L. Bettini, F. Damiani, and I. Schaefer. Implementing software product lines using traits. In *SAC*, pages 2096–2102. ACM, 2010.

[7] L. Bettini, F. Damiani, and I. Schaefer. Implementing Type-Safe Software Product Lines using Records and Traits. Technical Report RT 135, Dipartimento di Informatica, Università di Torino, 2011. Available at *http://www.di.unito.it/˜damiani/papers/tr-135-2011.pdf*.

[8] L. Bettini, F. Damiani, I. Schaefer, and F. Strocco. A prototypical Java-like language with records and traits. In *PPPJ*, pages 129–138. ACM, 2010.

[9] V. Bono, F. Damiani, and E. Giachino. On Traits and Types in a Java-like setting. In *TCS (Track B)*, volume 273 of *IFIP*, pages 367–382. Springer, 2008.

[10] G. Booch, J. Rumbaugh, and I. Jacobson. *The Unified Modeling Language User Guide*. Addison-Wesley, 1999.

[11] F. Damiani, J. Dovland, E. B. Johnsen, and I. Schaefer. A Proof System for Fine-Grained Reuse (version with Appendix). Technical Report RT 140, Dip. di Informatica, Università di Torino, 2011. Available at *http://www.di.unito.it/˜damiani/papers/tr-140-2011.pdf*.

[12] F. S. de Boer. A WP-calculus for OO. In W. Thomas, editor, *Proceedings of Foundations of Software Science and Computation Structure, (FOSSACS'99)*, volume 1578 of *LNCS*, pages 135–149. Springer, 1999.

[13] J. Dovland, E. B. Johnsen, O. Owe, and M. Steffen. Lazy behavioral subtyping. *Journal of Logic and Algebraic Programming*, 79(7):578–607, 2010.

[14] J. Dovland, E. B. Johnsen, O. Owe, and M. Steffen. Incremental reasoning with lazy behavioral subtyping for multiple inheritance. *Science of Computer Programming*, 76(10):915–941, 2011.

[15] S. Ducasse, O. Nierstrasz, N. Schärli, R. Wuyts, and A. P. Black. Traits: A mechanism for fine-grained reuse. *ACM Transactions on Programming Languages and Systems*, 28(2):331–388, 2006.

[16] C. A. R. Hoare. An Axiomatic Basis of Computer Programming. *Communications of the ACM*, 12:576–580, 1969.

[17] A. Igarashi, B. Pierce, and P. Wadler. Featherweight Java: A minimal core calculus for Java and GJ. *ACM Transactions on Programming Languages and Systems*, 23(3):396–450, 2001.

[18] E. B. Johnsen, O. Owe, and I. C. Yu. Creol: A type-safe object-oriented model for distributed concurrent systems. *Theoretical Computer Science*, 365(1–2):23–66, Nov. 2006.

[19] G. T. Leavens and D. A. Naumann. Behavioral subtyping, specification inheritance, and modular reasoning. Technical Report 06-20a, Department of Computer Science, Iowa State University, Ames, Iowa, 2006.

[20] L. Liquori and A. Spiwack. Extending FeatherTrait Java with interfaces. *Theoretical Computer Science*, 398(1-3):243–260, 2008.

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

[22] C. Luo and S. Qin. Separation logic for multiple inheritance. *ENTCS*, 212:27–40, April 2008.

[23] O. Nierstrasz, S. Ducasse, and N. Schärli. Flattening traits. *JOT*, 5(4):129–148, 2006.

[24] M. Nordio, C. Calcagno, P. Müller, and B. Meyer. A Sound and Complete Program Logic for Eiffel. In M. Oriol, editor, *TOOLS-EUROPE 2009*, volume 33 of *Lecture Notes in Business and Information Processing*, pages 195–214, 2009.

[25] M. Odersky. The Scala Language Specification, version 2.4. Technical report, Programming Methods Laboratory, EPFL, 2007.

[26] S. Owicki and D. Gries. An axiomatic proof technique for parallel programs I. *Acta Informatica*, 6(4):319–340, 1976.

[27] A. Poetzsch-Heffter and P. Müller. A programming logic for sequential Java. In S. D. Swierstra, editor, *8th European Symposium on Programming Languages and Systems (ESOP'99)*, volume 1576 of *LNCS*, pages 162–176. Springer, 1999.

[28] J. Reppy and A. Turon. A foundation for trait-based metaprogramming. In *Electronic proceedings of FOOL/WOOD 2006*, 2006.

[29] J. Reppy and A. Turon. Metaprogramming with traits. In *ECOOP*, volume 4609 of *LNCS*, pages 373–398. Springer, 2007.

[30] M. Schwerhoff. Verifying Scala traits. Semester Report, Swiss Federal Institute of Technology Zurich (ETH), Oct. 2010.

[31] C. Smith and S. Drossopoulou. *Chai*: Traits for Java-like languages. In *ECOOP*, volume 3586 of *LNCS*, pages 453–478. Springer, 2005.

[32] S. van Staden and C. Calcagno. Reasoning about multiple related abstractions with multistar. In *OOPSLA '10*, pages 504–519. ACM, 2010.