Traits: The Formal Model

Nathanael Schärli, Oscar Nierstrasz, Stéphane Ducasse,
Roel Wuyts and Andrew Black

Department of Computer Science and Engineering
OGI School of Science & Engineering
Oregon Health & Science University
20000 NW Walker Road
Beaverton, OR 97006-8921 USA

Technical Report Number CSE 02-013

also

Universität Bern
Institut für Informatik und angewandte Mathematik
Neubrückstrasse 10
CH-3012, Bern, Switzerland

Technical Report Number IAM-02-006

Revised: 24th February 2003

Traits: The Formal Model*

Nathanael Schärli, Oscar Nierstrasz, Stéphane Ducasse, Roel Wuyts, and Andrew Black

Software Composition Group, University of Bern, Switzerland
OGI School of Science & Engineering, Oregon Health and Science University
{schaerli, oscar, ducasse, wuyts}@iam.unibe.ch, black@cse.ogi.edu

Abstract. *Traits* are reusable units of behaviour that provide a level of structuring for object-oriented programs above the level of methods but below the level of classes. Structuring classes with single-inheritance alone can lead to duplicated code when different branches of the hierarchy need to use the same feature. Multiple-inheritance and mixins alleviate this problem, but lead to other difficulties in the face of evolution: changes to classes or mixins can break code lower in the hierarchy in unexpected ways.

Traits solve both problems by factoring out shared behaviour as sets of methods that do not depend on state. Trait composition is symmetric and conflicts must be managed explicitly; this means that changes in the components do not lead to unexpected side effects.

This paper presents a formal model of traits, and defines some basic properties of traits and classes. We also model the internal dependencies created by **self** and **super**-sends so that we can specify precisely when two classes are equivalent.

1 Introduction

Traits provide a mechanism that can be used to factor out shared behaviour from classes. The purpose of this paper is to provide a formal basis for understanding and reasoning about traits; it is not intended to be self-contained. A companion paper, which we will refer to as the "traits paper" [SDNB02], shows that traits overcome many difficulties with single and multiple inheritance, and with mixins. We have also reported on the use of an experimental implementation to refactor a part of the Smalltalk collections hierarchy [BSD02].

The semantics of single inheritance is well understood and it is described by several operational and denotational formalizations [GR83,CP89]. For conciseness, we do not repeat any details of these formalizations. Instead we focus on two contributions. First, we define traits and the various operators that can be used to combine traits into compound traits, and ultimately into classes. Second, we make precise the internal dependencies within a class that are created when methods send messages to **self** and **super**. This enables us to specify when

* −Id: model3.tex,v 1.40 2003/02/24 16:43:37 oscar Exp −

different ways of composing a class are equivalent from the perspective of provided and required services. Depending on the programmer's goals, equivalent classes might be regarded as legal refactorings, or as alternative views on a more abstract form of the program.

2 The Traits Language

While developing traits we have not focussed on issues of concrete syntax. In fact, because our initial implementation is part of the Smalltalk interactive development environment, we have been able to defer syntactic considerations almost entirely: traits are created and manipulated by the traits browser using method templates and menu selections. However, we have found that readers grasp the essentials of traits more quickly if they can see the various forms displayed as a grammar, so in the traits paper we presented an abstract syntax for the principal operations on traits [SDNB02, section 4]. Here we reproduce that grammar, and expand it to include constructs for method dictionaries, for the inheritance operation on classes, and for definitions that give names to classes and traits.

$$
\begin{array}{lll}
\text{method dictionary:} & D & ::= \{\} \mid \{x_1 \mapsto m_1, \ldots, x_n \mapsto m_n\} \\
\text{trait expression:} & T & ::= N_T \mid T + T \mid T - x \mid T[x \rightarrow y] \\
\text{trait definition:} & T_{Def} & ::= N_T = D \mid N_T = D \textsf{ with } T \\
\text{fields:} & F & ::= \{\} \mid \{f_1, f_2, \ldots, f_n\} \\
\text{class expression:} & C & ::= \textsf{nil} \mid N_C \mid F \textsf{ in } D \textsf{ extends } N_C \\
 & & \quad \mid \; F \textsf{ in } D \textsf{ with } T \textsf{ extends } N_C \\
\text{class definition:} & C_{Def} & ::= N_C = C
\end{array}
$$

D denotes a *method dictionary*, which is either empty or contains a finite number of *services* (*i.e.*, bindings of labels to methods) in which all of the labels x_i are distinct. T denotes a *trait expression*, which can be a named trait N_T, the symmetric composition $T + T$ of two traits, or a trait that is subject to an exclusion $T - x$ or an alias $T[x \rightarrow y]$. A *trait definition* binds a trait name N_T to one of two forms: D, a method dictionary, or $N_T = D$ **with** T, a method dictionary that adds methods to (and possibly overrides methods from) a trait.

F is a possibly empty set of variable (or *field*) names; it is not important for our purposes whether these are instance variables, class variables, or static variables. C denotes a class, which can be nil, a named class N_C, or a conventional class F **in** D **extends** N_C, which consists of some fields F, a method dictionary D, and a named superclass N_C. In addition, a class can be *structured* F **in** D **with** T **extends** N_C; the structured class acquires methods from the trait T augmented (and possibly overridden) by those from the explicit method dictionary D. Finally, a *class definition* C_{Def} is simply a binding of a class name N_C to a class C.

Since it is context free, this abstract syntax does not capture several context conditions. These are that a name N_T cannot be used as a trait expression until it has been defined (and similarly for N_C). Also, the form F **in** D places the fields

defined in F in scope within the method bodies m_i that appear in D. Conversely, *no* fields are in scope in D in the forms $N_T = D$ and $N_T = D$ **with** T.

The syntax *does* capture some important restrictions that the traits browser imposes for reasons of human engineering. The **with** keyword, which allows method overriding, is second class. It can appear only in the definition of a named trait or class, and only with a literal method dictionary on the left hand side: these restrictions help to ensure that all overriding of services in a trait is explicit.

3 A Model for Traits

3.1 Informal Introduction

Because it is not important for the semantics of traits, we do not represent the internal structure of methods in our model. Instead, we model methods as atomic values in a set $\mathcal{M}$, augmented with top and bottom elements to create a flat lattice; $\bot$ represents "no method" and $\top$ stands for a "method conflict". This also means that our model is independent of any particular programming language.

Methods should be thought of as implementations, or code sequences. All we assume about methods is that we can ascertain when they may send messages to instances of their class. Such message sends are of two kinds: self-sends and super-sends. Whereas self-sends employ the usual message lookup that starts in the class of the current instance, super-sends cause the message lookup to start in the superclass of the class that defines the method containing the super-send.

Traits are essentially sets of bindings of *labels* to methods; the labels name the methods. We call these bindings *services*. Traits both *provide* services (*i.e.*, the methods implemented) and *require* services (*i.e.*, those that are invoked by self- and super-sends). When we say that a class or a trait provides a service with a particular label we mean that a binding of that label to a method exists. When we say that a service with a particular label is required, we mean that a binding of that label to a method needs to exist.

We introduce *summation* ($+$), *exclusion* ($-$), *aliasing* ($\rightarrow$) and *overriding* ($\triangleright$) as mechanisms for composing and modifying traits. When traits are composed with $+$, labels that are bound to different methods will *conflict*; we represent this by binding the label to $\top$ in the sum. The $\triangleright$ operation is intended to be used to override these bindings to $\top$ with proper methods.

Traits are subordinate to classes in several ways. Unlike classes, traits bind neither **self** nor **super**. Classes can contain fields, and methods in classes can refer to those fields, so it is possible for classes to provide *accessor* methods (also known as*setters* and *getters*). In contrast, traits do *not* contain fields, and as a consequence methods in traits *cannot* refer to fields. However, because the requirements of a trait method can be met by accessor methods, a trait method can access state indirectly.

A class is built up from traits, methods, and a single superclass using single inheritance. If the traits conflict, the class can eliminate the conflict by overriding

the conflicting methods. **super** is bound only at the point that a class is composed, and **self** refers to an instance of a composed class.

The labels *required* by a class consist of the set of self-sends made by all the methods reachable by the super-send chain, minus the set of labels of the *provided* services. A class is *concrete* if all its requirements are fulfilled, otherwise it is *abstract*. A class is *well-founded* if the super-sends are properly bound; note that a well-founded class can still be abstract. Two classes are *equivalent* if they provide the same set of services, and all the methods reachable by sequences of self- and super-sends are the same.

In the remainder of this section we make this model formally precise. In section 4 we consider several examples from the traits paper, and show how they are represented in the model.

3.2 Methods

The primitive elements of our model are the following disjoint sets:

- $\mathcal{M}$, a countable set of *methods*,
- $\mathcal{L}$, a countable set of method *labels*, and
- $\mathcal{F}$, a countable set of *field* labels.

Because of the context restriction mentioned in section 2, trait methods cannot access fields directly. In contrast, class methods *can* access fields, and the method dictionaries of classes can define accessor methods that can be used by trait methods to access fields indirectly. We do not make this distinction explicit in the model by partitioning $\mathcal{M}$. Instead, it will be sufficient to model methods as atomic elements, and not to consider their internal structure. For instance, we make no assumption on how equality of methods is tested in an actual language: $m_1 = m_2$ might be equality of strings, of syntax trees, of bytecode, or of something else.

We are, however, interested in which other methods are *required* by a given method. We therefore model:

- $\mathit{selfSends} : \mathcal{M} \to 2^{\mathcal{L}}$, the set of labels used in self-sends, and
- $\mathit{superSends} : \mathcal{M} \to 2^{\mathcal{L}}$, the set of labels used in super-sends.

We extend *selfSends* and *superSends* to sets of methods by union:

- $\mathit{selfSends} : 2^{\mathcal{M}} \to 2^{\mathcal{L}}$

$$\mathit{selfSends}(\mu) = \bigcup_{m \in \mu} \mathit{selfSends}(m)$$

- $\mathit{superSends} : 2^{\mathcal{M}} \to 2^{\mathcal{L}}$

$$\mathit{superSends}(\mu) = \bigcup_{m \in \mu} \mathit{superSends}(m)$$

Note that the precise definitions of *selfSends* and *superSends* are language-dependent. For example, consider the method

```
fasten
    — anObject — self hook.
    anObject := self.
    anObject button.
    self class new clip.
```

It is clear that hook should be recognized as a self-send. What about button? A little local static anaylsis ought to be able to show that this is also a self-send. Is clip a self-send? We avoid answering this question! clip is a requirment on the class, but whether it is recognized as such depends on how thorough an analysis is carried out by the tools. We assume only that *selfSends*(m) provides a *lower bound* on the set of messages that an object might send to instances of its class while executing m. Other messages might be sent through indirect paths that are hard to discover.

We extend the set of methods $\mathcal{M}$ to a flat lattice $\mathcal{M}^\star$, with $\bot \sqsubset m \in \mathcal{M} \sqsubset \top$, and in which all other elements are incomparable. We will use $\bot$ to represent *undefined* and $\top$ to represent a method *conflict*. Thus, the *least upper bound* or *join* operator $\sqcup$ for $\mathcal{M}^\star$ is as shown:

$\sqcup$	$\bot$	m_1	m_2	$\top$
$\bot$	$\bot$	m_1	m_2	$\top$
m_1	m_1	m_1	$\top$	$\top$
m_2	m_2	$\top$	m_2	$\top$
$\top$	$\top$	$\top$	$\top$	$\top$

where $m_1 \neq m_2$

@@ add the lattice digram to the right of this table @@

Definition 1. *A* service *is a partial function binding a single method label to a particular method. We use the notation:*

$$x \mapsto m$$

for the service that binds the label $x \in \mathcal{L}$ *to the method* $m \in \mathcal{M}$*, and*

$$x \mapsto \top$$

to indicate that label x *is in conflict.*

3.3 Traits

Definition 2. *A* trait*,* $t \in \mathcal{T}$*, is a finite set of services, i.e., a function,*

$$t : \mathcal{L} \rightarrow \mathcal{M}^\star$$

where $\{l \in \mathcal{L} \mid t(l) \neq \bot\}$ *is a finite set.*

Since traits are finite, we will always specify them extensionally, listing only bindings to elements $\mathcal{M}$ and $\top$. For example,

$$t = \{x \mapsto m_1, y \mapsto m_2, z \mapsto \top\}$$

defines a trait t that maps label x to method m_1, label y to label m_2, and for which label z has a conflict.

By convention, *selfSends* and *superSends* of $\top$ and $\bot$ are all $\emptyset$. We extend *selfSends* and *superSends* to traits in the obvious way:

- $\mathit{selfSends} : \mathcal{T} \rightarrow 2^{\mathcal{L}}$

$$\mathit{selfSends}(t) \stackrel{\text{def}}{=} \mathit{selfSends}(t(\mathcal{L}))$$

- $\mathit{superSends} : \mathcal{T} \rightarrow 2^{\mathcal{L}}$

$$\mathit{superSends}(t) \stackrel{\text{def}}{=} \mathit{superSends}(t(\mathcal{L}))$$

Since traits are just finite mappings, two traits are equal when these mappings are equal, that is, when equal labels map to equal methods.

Definition 3. *The* conflicts*, $\mathit{conflicts} : \mathcal{T} \rightarrow 2^{\mathcal{L}}$, of a trait t are defined by:*

$$\mathit{conflicts}(t) \stackrel{\mathit{def}}{=} \{l \mid t(l) = \top\}$$

Definition 4. *A* method dictionary *d is a* conflict-free *trait, that is, a trait d such that $\mathit{conflicts}(d) = \emptyset$.*

Conflict-free traits model the literal method dictionaries D of the traits language (see section 2).

Definition 5. *The* provided *labels, $\mathit{provided} : \mathcal{T} \rightarrow 2^{\mathcal{L}}$, of a trait t are defined as:*

$$\mathit{provided}(t) \stackrel{\mathit{def}}{=} \{l \in \mathcal{L} \mid t(l) \in \mathcal{M}\}$$

i.e., the set of all labels that t does not map to $\bot$ or $\top$.

Definition 6. *The* required *labels, $\mathit{required} : \mathcal{T} \rightarrow 2^{\mathcal{L}}$, of a trait t are defined as:*

$$\mathit{required}(t) \stackrel{\mathit{def}}{=} \mathit{selfSends}(t) \setminus \mathit{provided}(t)$$

Notice that the required labels of a trait do not consider super-sends. However, when we compose classes from traits we must take super-sends into account in order to determine the required labels of the class (cf. Definition 17).

3.4 Trait Operations

Traits can either be specified directly, as lists of services, or they may be composed from other traits, by means of *summation* ($+ : \mathcal{T} \times \mathcal{T} \to \mathcal{T}$), *exclusion* ($- : \mathcal{T} \times \mathcal{L} \to \mathcal{T}$), *aliasing* ($_[_ := _] : \mathcal{T} \times \mathcal{L} \times \mathcal{L} \to \mathcal{T}$), and *overriding* ($\triangleright : \mathcal{T} \times \mathcal{T} \to \mathcal{T}$).

Definition 7. Summation *composes two traits, forming the union of non-conflicting bindings and "disabling" the conflicting bindings:*

$$(t_1 + t_2)(l) \stackrel{def}{=} t_1(l) \sqcup t_2(l)$$

For example:

$$\{a \mapsto m_1, b \mapsto m_2, c \mapsto m_3\} + \{a \mapsto m_1, b \mapsto m_3\} = \{a \mapsto m_1, b \mapsto \top, c \mapsto m_3\}$$

Proposition 1. *Symmetric composition is associative and commutative.*

Proof. Immediate from the definition, since the *join* operator $\sqcup$ is associative and commutative.

Definition 8. Exclusion *removes a binding from a trait:*

$$(t - l)(x) \stackrel{def}{=} \begin{cases} \bot & \textit{if } x = l \\ t(x) & \textit{otherwise} \end{cases}$$

For example:

$$\{a \mapsto m_1, b \mapsto \top\} - b = \{a \mapsto m_1\}$$

Definition 9. Aliasing *introduces an additional name for an existing method:*

$$t[x{:=}y](l) \stackrel{def}{=} \begin{cases} t(y) & \textit{if } l = x \\ t(l) & \textit{otherwise} \end{cases}$$

For example:

$$\{a \mapsto m_1, b \mapsto m_2\}[c{:=}b] = \{a \mapsto m_1, b \mapsto m_2, c \mapsto m_2\}$$

These three operations model the $+$, $-$ and $\to$ operations from the language syntax of section 2.

When composing classes or traits, we may need to replace conflicts by proper methods. For this reason we introduce the $\triangleright$ operator.

Definition 10. *A trait t_1 may* replace *some of the bindings in another trait t_2, as follows:*

$$(t_1 \triangleright t_2)(l) \stackrel{def}{=} \begin{cases} t_2(l) & \textit{if } t_1(l) = \bot \\ t_1(l) & \textit{otherwise} \end{cases}$$

The $\triangleright$ operator models the **with** syntax of the traits language; the trait d **with** t is modeled by $d \triangleright t$. Because $\triangleright$ is asymmetric, in our concrete traits language its use is limited to definitions of named traits and named classes in which literal method definitions appear on the left hand side (see the discussion in Section 2).

3.5 Classes

Whereas traits are just sets of services without any context, a class binds **self** and **super** for its methods. In addition, a concrete class must provide all its required methods. Because the method referred to by a super-send depends on which methods are defined by the various superclasses, to define the meaning of "required" for a class we must model the entire inheritance chain of the class. We do this by representing a class as a *sequence* of method dictionaries.

Definition 11. *A* class, *$c \in \mathcal{C}$, is either the empty class,* nil*, or a sequence $\langle \phi, t \rangle \cdot c'$, where $\phi \subset \mathcal{F}$, $t \in \mathcal{T}$, and $c' \in \mathcal{C}$.*

If we assume that s_m is the model for s, then the conventional class

$$\phi \ \textsf{\textbf{in}}\ d \ \textsf{\textbf{extends}}\ s$$

is modeled by $\langle \phi, d \rangle \cdot s_m$, and the structured class

$$\phi \ \textsf{\textbf{in}}\ d \ \textsf{\textbf{with}}\ t \ \textsf{\textbf{extends}}\ s$$

is modeled by $\langle \phi, d \rhd t \rangle \cdot s_m$.

Running Example We will refer to the following example throughout the remainder of this section.

$$\begin{aligned} c = & \langle \{i\}, \{a \mapsto m_6, x \mapsto m_1, y \mapsto m_2\} \rangle \\ & \cdot \langle \{j\}, \{b \mapsto m_7, x \mapsto m_3, y \mapsto m_4\} \rangle \\ & \cdot \langle \emptyset, \{z \mapsto m_5\} \rangle \\ & \cdot \textsf{nil} \end{aligned}$$

where

$$\begin{array}{ll} \mathit{selfSends}(m_1) = \{a, y\} & \mathit{superSends}(m_1) = \{x\} \\ \mathit{selfSends}(m_2) = \emptyset & \mathit{superSends}(m_2) = \emptyset \\ \mathit{selfSends}(m_3) = \{b\} & \mathit{superSends}(m_3) = \emptyset \\ \mathit{selfSends}(m_4) = \{w\} & \mathit{superSends}(m_4) = \{z\} \\ \mathit{selfSends}(m_5) = \{a\} & \mathit{superSends}(m_5) = \emptyset \\ \mathit{selfSends}(m_6) = \emptyset & \mathit{superSends}(m_6) = \emptyset \\ \mathit{selfSends}(m_7) = \emptyset & \mathit{superSends}(m_7) = \emptyset \end{array}$$

For conciseness, we will in future omit the trailing $\cdot$nil from all examples.

Definition 12. *The* incremental method dictionary*, $\mathit{delta}(c)$, of a class c, is:*

$$\mathit{delta}(c) \stackrel{\mathit{def}}{=} \begin{cases} \{\} & \mathit{if}\ c = \textsf{nil} \\ d & \mathit{if}\ c = \langle \phi, d \rangle \cdot c' \end{cases}$$

Definition 13. *The* superclass*, $\mathit{super}(c)$, of c is:*

$$\mathit{super}(c) \stackrel{\mathit{def}}{=} \begin{cases} \textsf{nil} & \mathit{if}\ c = \textsf{nil} \\ c' & \mathit{if}\ c = \langle \phi, d \rangle \cdot c' \end{cases}$$

Definition 14. *The* flattening *of a class c, $\mathit{flatten}(c)$, is the $\triangleright$ composition of the method dictionaries in its inheritance chain:*

$$\mathit{flatten}(c) \stackrel{def}{=} \begin{cases} \{\} & \text{if } c = \mathsf{nil} \\ d \triangleright \mathit{flatten}(c') & \text{if } c = \langle \phi, d \rangle \cdot c' \end{cases}$$

Definition 15. *The* method lookup*, $c \gg x$, of a label x in a class c is:*

$$c \gg x \stackrel{def}{=} \mathit{flatten}(c)(x)$$

For our running example:

$$\begin{aligned} \mathit{delta}(c) &= \{a \mapsto m_6, x \mapsto m_1, y \mapsto m_2\} \\ \mathit{super}(c) &= \langle \{b\}, \{b \mapsto m_7, x \mapsto m_3, y \mapsto m_4\} \rangle \cdot \langle \emptyset, \{z \mapsto m_5\} \rangle \\ \mathit{delta}(\mathit{super}(c)) &= \{b \mapsto m_7, x \mapsto m_3, y \mapsto m_4\} \\ \mathit{super}(\mathit{super}(c)) &= \langle \emptyset, \{z \mapsto m_5\} \rangle \\ \mathit{flatten}(c) &= \{a \mapsto m_6, b \mapsto m_7, x \mapsto m_1, y \mapsto m_2, z \mapsto m_5\} \\ \mathit{flatten}(\mathit{super}(c)) &= \{b \mapsto m_7, x \mapsto m_3, y \mapsto m_4, z \mapsto m_5\} \\ \mathit{flatten}(\mathit{super}(\mathit{super}(c))) &= \{z \mapsto m_5\} \\ c \gg y &= m_2 \\ \mathit{super}(c) \gg x &= m_3 \\ \mathit{super}(\mathit{super}(c)) \gg x &= \bot \end{aligned}$$

Definition 16. *The* provided *labels, $\mathit{provided} : \mathcal{C} \to 2^{\mathcal{L}}$, of a class c are defined as:*

$$\mathit{provided}(c) \stackrel{def}{=} \{l \in \mathcal{L} \mid c \gg l \in \mathcal{M}\}$$

i.e., the set of all labels that $\mathit{flatten}(c)$ does not map to $\bot$ or $\top$.

Definition 17. *A class c is* well-founded *if and only if all super-sends in its inheritance chain are bound, i.e., if $\mathit{superSends}(\mathit{delta}(c)) \subseteq \mathit{provided}(\mathit{super}(c))$ and $\mathit{super}(c)$ is well-founded.* nil *is well-founded by convention.* [1]

For a particular programming language, a class that is not well-founded may generate run-time errors, or compile-time errors, depending on the philosphy of its designers.

In the running example:

$$\mathit{provided}(c) = \{a, b, x, y, z\}$$

c is well-founded since the super-send to x in m_1 is bound in $\mathit{super}(c)$.

[1] This definition is slightly more restrictive than necessary; in fact, it does no harm for the superclass to contain a method that has an unbound supersend, provided that this method is not reachable. Of course, such a method is of no practical use.

3.6 Superclass Reachability and Class Equivalence

The *required* labels of a class are more subtle than those of a trait, because we must consider not only self-sends, but also the entire inheritance chain of super-sends. In the running example c, it is crucial to capture that the fact that the accessor method b is required (and fulfilled) because m_3 is reachable through a chain of super-sends, whereas w is not required because m_4 is not reachable.

Thus, to determine which methods are reachable, we see that we must consider not only the methods bound to the provided labels, but methods that may be reached by super-sends from the corresponding methods, methods that may be reached by super-sends from those methods, and so on. We introduce the notation $c{\uparrow}x_1x_2...x_n = \langle m, c'\rangle$ to mean that it is possible for the method bound to x_1 in class c to perform a super-send to x_2, and for the method bound to x_2 in this context to perform a supersend .... to x_n. If this occurs, then x_n will be bound to method m obtained from c'. For example, $c{\uparrow}x = \langle m_1, c\rangle$, since the service $x \mapsto m_1$ occurs in $\mathit{delta}(c)$, but $c{\uparrow}b = \langle m_7, \mathit{super}(c)\rangle$. For convenience, we introduce the notation $c{\upharpoonright}\bar{x}$, which returns just the method reachable by $\bar{x}$ without its class, so $c{\upharpoonright}x = m_1$ and $c{\upharpoonright}b = m_7$.

Reachability for longer chains must take super-sends into account, so $c{\uparrow}xx = \langle m_3, \mathit{super}(c)\rangle$, since m_3 is reachable by the super-send chain xx. On the other hand, $c{\uparrow}xy = \bot$, since $y \notin \mathit{superSends}(m_1)$. Note that although m_4 is not reachable by any super-send chain starting from c, it is reachable from $\mathit{super}(c)$.

Definition 18. *$c{\uparrow}\bar{x}$, where $c \in \mathcal{C}$ and $\bar{x} \in \mathcal{L}^+$, is defined recursively, as follows:*

$$
\begin{aligned}
\mathsf{nil}{\uparrow}x &\stackrel{def}{=} \langle \bot, \mathsf{nil}\rangle \\
c{\uparrow}x &\stackrel{def}{=} \begin{cases} \langle m, c\rangle & \text{if } m = \mathit{delta}(c)(x) \in \mathcal{M} \\ \mathit{super}(c){\uparrow}x & \text{otherwise} \end{cases} \\
c{\uparrow}\bar{x}y &\stackrel{def}{=} \begin{cases} \mathit{super}(c'){\uparrow}y & \text{if } c{\uparrow}\bar{x} = \langle m, c'\rangle \text{ and } y \in \mathit{superSends}(m) \\ \langle \bot, \mathsf{nil}\rangle & \text{otherwise} \end{cases}
\end{aligned}
$$

Definition 19.

$$c{\upharpoonright}\bar{x} \stackrel{def}{=} m, \text{ where } \exists c' \text{ such that } c{\uparrow}\bar{x} = \langle m, c'\rangle$$

Theorem 1. *For any class c and any single message x, $c{\upharpoonright}x = c \gg x$*

Proof. *By induction on the length of the super-send chain. The base case where $c = \mathsf{nil}$ is trivial. The induction step has two cases, where x is bound in delta(c) and where it is not.*

Definition 20. *A method $m \in \mathcal{M}$ is* reachable *from class c, if $\exists\bar{x} \in \mathcal{L}^+$ such that $m = c{\upharpoonright}x$.*

Definition 21. *The set of methods* reachable *from a class c, is:*

$$\mathit{reachable}(c) \stackrel{def}{=} \{c{\upharpoonright}\bar{x} \mid \bar{x} \in \mathcal{L}^+\} \setminus \{\bot\}$$

In the running example, $\mathit{reachable}(c) = \{m_1, m_2, m_3, m_5, m_6, m_7\}$.

Definition 22. *The set of* required *labels, $\mathit{required}(c)$, of a class c is:*

$$\mathit{required}(c) \stackrel{def}{=} \mathit{selfSends}(\mathit{reachable}(c)) \setminus \mathit{provided}(c)$$

Note that the required labels include the self-sends, not only of the provided methods, but also of all methods reachable through super-sends.

Definition 23. *A class, $c \in \mathcal{C}$, is* concrete *if $\mathit{required}(c) = \emptyset$. A class that is not concrete is* abstract*.*

In the example, $\mathit{required}(c) = \{a, b, y\} \setminus \{a, b, x, y, z\} = \emptyset$, so c is concrete. In contrast, $\mathit{required}(\mathit{super}(c)) = \{b, w\} \setminus \{b, x, y, z\} = \{w\}$, so $\mathit{super}(c)$ is abstract. Similarly, $\mathit{required}(\mathit{super}(\mathit{super}(c))) = \{a\} \setminus \{y\} = \{a\}$, so $\mathit{super}(\mathit{super}(c))$ is also abstract.

Two classes are equivalent if exactly the same methods are reachable by the same super-send chains.

Definition 24. *A class c is* equivalent *to a class c', $c \equiv c'$, iff:*
$\mathit{provided}(c) = \mathit{provided}(c')$ and $\forall \bar{x} \in \mathcal{L}^+, c\rceil\bar{x} = c'\rceil\bar{x}$.

Suppose that:

$$c' = \langle\{a\}, \{a \mapsto m_6, x \mapsto m_1, y \mapsto m_2, z \mapsto m_5\}\rangle \cdot \langle\{b\}, \{b \mapsto m_7, x \mapsto m_3\}\rangle$$

Then $c \equiv c'$ since

$$\begin{array}{rcl}
\mathit{provided}(c) = & \mathit{provided}(c') & = \{a, b, x, y, z\} \\
c\rceil x = & c'\rceil x & = m_1 \\
c\rceil y = & c'\rceil y & = m_2 \\
c\rceil a = & c'\rceil a & = m_6 \\
c\rceil b = & c'\rceil a & = m_7 \\
c\rceil z = & c'\rceil z & = m_5 \\
c\rceil xx = & c'\rceil xx & = m_3
\end{array}$$

and no other methods are reachable.

4 Some Examples

In this section, we give the formalizations of some of the examples presented in the traits paper [SDNB02]. For conciseness, we omit the methods to which labels are bound except when there is a method conflict. For example, rather than writing $\{a \mapsto m_1, b \mapsto m_2, c \mapsto \top\}$ we simply write $\{a, b, c \mapsto \top\}$.

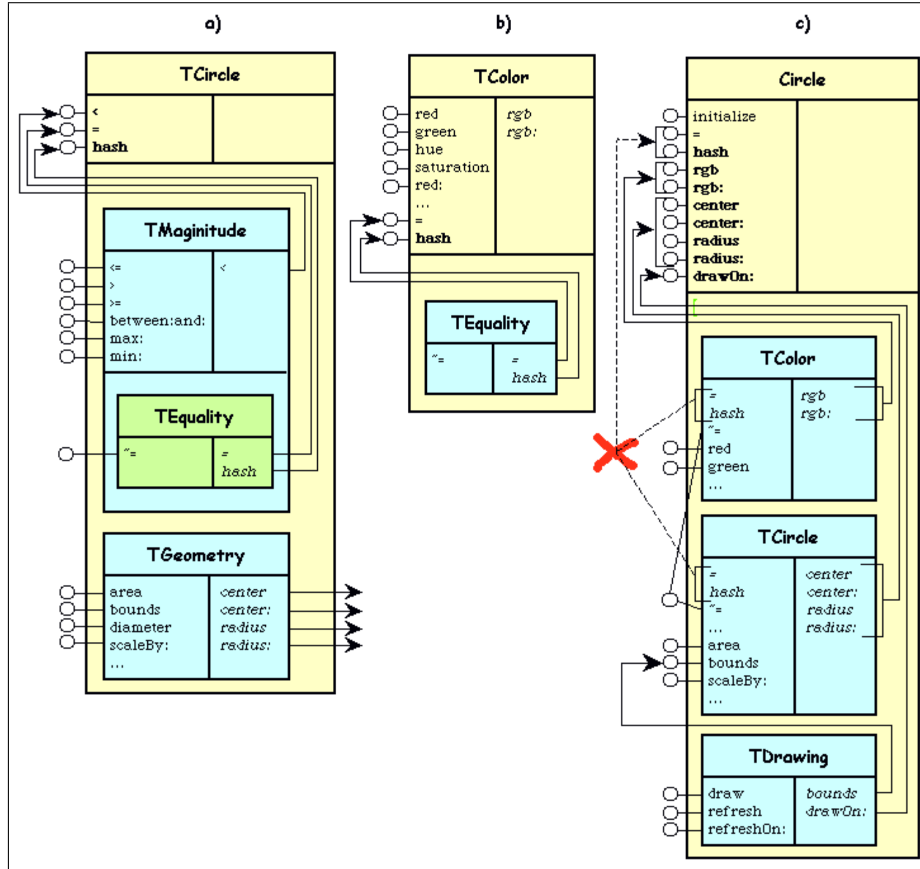

Fig. 1. (a) shows how a trait TCircle is composed from a trait TGeometry and a nested trait TMagnitude, which is again composed from the trait TEquality. Note that the provided services of the nested traits are propagated to the composite trait (*e.g.*, max:, ∼=, and area), and similarly, the unsatisfied requirements of the nested traits (*e.g.*, center and radius:) are turned into required methods of the composite trait. In (b), we again use the trait TEquality to specify the comparison behaviour of the trait TColor. (c) shows how a class Circle is specified by composing the traits TCircle, TColor, and TDrawing.

$$
\begin{aligned}
\mathit{TEquality} &= \{\sim=\}\\[1ex]
\mathit{TDrawing} &= \{\mathit{draw}, \mathit{refresh}, \mathit{refreshOn:}\}\\[1ex]
\mathit{TGeometry} &= \{\mathit{area}, \mathit{bounds}, \mathit{diameter}, \mathit{scaleBy:}, \ldots\}\\[1ex]
\mathit{TMagnitude} &= \{<=, >, >=, \mathit{between:and:}, \mathit{max:}, \mathit{min:}\} \textsf{ with } \mathit{TEquality}\\[1ex]
\mathit{TColor} &= \{\mathit{red}, \mathit{green}, \mathit{blue}, \mathit{hue}, \mathit{saturation}, \mathit{red:}, \ldots, =, \mathit{hash}\}\\
&\quad \textsf{with } \mathit{TEquality}\\[1ex]
\mathit{TCircle} &= \{<, =, \mathit{hash}\} \textsf{ with } (\mathit{TMagnitude} + \mathit{TGeometry})\\[1ex]
\mathit{Circle} &= \langle\{\mathit{center}, \mathit{radius}, \mathit{rgb}\},\\
&\quad \{\mathit{center}, \mathit{center:}, \mathit{radius}, \mathit{radius:}, \mathit{rgb}, \mathit{rgb:}, \mathit{initialize}, =, \mathit{hash}, \mathit{drawOn:}\}\\
&\quad \textsf{with } (\mathit{TColor}[\mathit{colorEquals:} \to =][\mathit{colorHash} \to \mathit{hash}]\\
&\qquad + \mathit{TCircle}[\mathit{circleEquals:} \to =][\mathit{circleHash} \to \mathit{hash}]\\
&\qquad + \mathit{TDrawing})\rangle\\
&\quad \textsf{extends } \mathit{Object}
\end{aligned}
$$

Fig. 2. Formal definition of the classes and traits shown in figure 1.

4.1 Graphical objects

In Section 3 of the traits paper [SDNB02, figure 5], we presented an example that uses traits to represent graphical objects, such as circles and rectangles, in a structured and reusable way. For ease of reference, the figure is reproduced here as figure 1. We now show how the traits and classes given in figure 1 are expressed in our model.

Figure 2 presents formalizations of all the traits and classes that are involved in this example. The first three traits *TEquality*, *TDrawing*, *TGeometry* are "primitive traits", which means that they do not make use of any other traits. None of these traits is complete, because they all send messages that do not correspond to provided services. In fact, the required labels of these traits are:

$$
\begin{aligned}
\mathit{required}(\mathit{TEquality}) &= \{=\}\\
\mathit{required}(\mathit{TDrawing}) &= \{\mathit{bounds}, \mathit{drawOn:}\}\\
\mathit{required}(\mathit{TGeometry}) &= \{\mathit{center}, \mathit{radius}, \mathit{radius}\}
\end{aligned}
$$

The trait *TMagnitude* uses the trait *TEquality*, and implements additional selectors such as $<=$, $<$, $>=$, *etc.*. Note that the trait *TMagnitude* does not provide the service $=$ required by the trait *TEquality*. This requirement is therefore propagated to the trait *TMagnitude*. Since *TMagnitude* additionally requires the service $<$, the complete set of requirements is $\mathit{required}(\mathit{TMagnitude}) = \{<, =\}$.

The trait *TColor* also uses the trait *TEquality*, but unlike *TMagnitude*, it implements a service $=$, which satisfies the requirement of *TEquality*. In addition,

TColor also implements some color methods such as *red*, *green*, *red:*, *etc.*, which require the services *rgb* and *rgb:*.

The trait *TCircle* is the composition of the traits *TMagnitude* and *TGeometry*. In addition, it implements the methods $<, =,$ *hash* and thereby satisfies all the requirements for *TMagnitude*. However, it does not satisfy any of the requirements of *TGeometry* and therefore, $\mathit{required}(\mathit{TCircle}) = \mathit{required}(\mathit{TGeometry})$.

Finally, we define the class *Circle*, making use of the traits *TColor*, *TCircle*, *TDrawing*. This means that we combine these three traits by using the symmetric composition operator $+$. Since *TColor* and *TCircle* provide different methods for $=$ and *hash*, this means that a conflict arises:

$$\mathit{conflicts}(\mathit{TColor} + \mathit{TCircle} + \mathit{TDrawing}) = \{=, \mathit{hash}\}$$

The class *Circle* resolves this conflict by implementing methods for $=$ and *hash*, which replace the conflicting methods. In order to allow these methods to access the methods from *TColor* and *TCircle*, we use aliasing to make these methods available in *Circle* under unambiguous names.

Note that the class *Circle* satisfies all the requirements of the traits that it uses by defining methods that access its fields *center*, *radius* and *rgb*.

4.2 Read-write synchronization

In Section 5.2 of the traits paper, we describe how we can build a trait that implements simple read-write synchronization in a reusable way. The problem assumes the existence of two classes, A and B that are not related by inheritance and provide unsynchronized *read* and *write* methods.

$$A = \langle \{\}, \{\mathit{read}, \mathit{write}, \ldots\} \rangle \cdot A'$$
$$B = \langle \{\}, \{\mathit{read}, \mathit{write}, \ldots\} \rangle \cdot B'$$

Our task is to create synchronized subclasses *SyncA* and *SyncB*. This can be accomplished by writing a trait *TSyncReadWrite*, that implements a simple synchronization strategy.

$$\mathit{TSyncReadWrite} = \{\mathit{read}, \mathit{write}, \mathit{acquireLock}, \mathit{releaseLock}\}$$

The methods *read* and *write* are implemented as shown in Figure 1 of the traits paper:

```
read
    self acquireLock.
    value := super read.
    self releaseLock.
    ↑ value

write
    self acquireLock.
    value := super write.
    self releaseLock.
```

```
↑ value
```

This means that the method *read* first calls the method *acquireLock*, then it calls the unsynchronized *read* method by using a super-send, and finally it calls the method *releaseLock*. The method *write* is similar. Since both *acquireLock* and *releaseLock* need to access a semaphore, the requirements and super-sends of *TSyncReadWrite* are as follows.

$$required(TSyncReadWrite) = \{semaphore\}$$
$$superSends(TSyncReadWrite) = \{read, write\}$$

Now we can use this trait to create subclasses *SyncA* of *A* and *SyncB* of *B*.

$$SyncA = \langle\{semaphore\}, \{semaphore\} \triangleright TSyncReadWrite\rangle \cdot A$$
$$SyncB = \langle\{semaphore\}, \{semaphore\} \triangleright TSyncReadWrite\rangle \cdot B$$

Since the superclasses *A* and *B* provide methods labeled *read* and *write*, the classes *SyncA* and *SyncB* are well-formed. And because *SyncA* and *SyncB* both provide an accessor method labeled *semaphore* (and the corresponding field), they are also concrete.

5 Summary

This paper has introduced a formal model for the mechanism described in the "traits paper" [SDNB02]. It has also formalized the notions of *required* and *provided* methods in an inheritance chain, and has used these notions to explain what is meant by a well-formed class and a concrete class. Notice that these definitions are in fact independent of the trait mechanism, and can be applied to any single-inheritance language.

References

[BSD02] Andrew Black, Nathanael Schärli, and Stéphane Ducasse. Applying traits to the Smalltalk collection hierarchy. Technical Report IAM-02-007, Institut für Informatik, Universität Bern, Switzerland, November 2002. Also available as Technical Report CSE-02-014, OGI School of Science & Engineering, Beaverton, Oregon, USA.

[CP89] William Cook and Jens Palsberg. A denotational semantics of inheritance and its correctness. In *Proceedings OOPSLA '89, ACM SIGPLAN Notices*, pages 433–443, October 1989. Published as Proceedings OOPSLA '89, ACM SIGPLAN Notices, volume 24, number 10.

[GR83] Adele Goldberg and David Robson. *Smalltalk 80: the Language and its Implementation.* Addison Wesley, Reading, Mass., May 1983.

[SDNB02] Nathanael Schärli, Stéphane Ducasse, Oscar Nierstrasz, and Andrew Black. Traits: Composable units of behavior. Technical Report IAM-02-005, Institut für Informatik, Universität Bern, Switzerland, November 2002. Also available as Technical Report CSE-02-014, OGI School of Science & Engineering, Beaverton, Oregon, USA.