Class Refinement and Interface Refinement in Object-Oriented Development

Anna Mikhajlova*
Turku Centre for Computer Science, Finland

Emil Sekerinski†
Åbo Akademi University, Finland

Abstract

Constructing new classes from existing ones by inheritance or subclassing is a characteristic feature of object-oriented development. Imposing semantic constraints on subclassing allows us to ensure that behaviour of superclasses is preserved or refined in their subclasses. This paper defines the class refinement relation which captures these semantic constraints. The class refinement relation is based on algorithmic and data refinement supported by the underlying formalism of Refinement Calculus. Class refinement is generalized to interface refinement, which takes place when a change in user requirements causes interface changes of classes designed as refinements of other classes. We formalize the interface refinement relation and present rules for refinement of clients of the classes involved in this relation.

1 Introduction

We demonstrate how formal methods, in particular, Refinement Calculus of Back, Morgan, and Morris [3, 15, 16], can be used for constructing more reliable object-oriented programs.

A characteristic feature of object-oriented program development is a uniform way of structuring all stages of the development by classes. The programming notation of Refinement Calculus is very convenient for describing object-oriented development because it allows us to specify classes at various abstraction levels. The specification language we use is based on monotonic predicate transformers, has class constructs, supports subclassing and subtype polymorphism. Besides usual imperative statements, the language includes specification statements which may appear in method bodies of classes leading to abstract classes. One of the main benefits offered by this language is that all development stages can be described in a uniform way starting with a simple abstract specification and resulting in a concrete program.

Another characteristic feature of object-oriented program development is constructing new classes from existing classes by inheritance, or subclassing. However, when a subclass overrides some methods of its superclass, there are no guarantees that its instances will deliver the same or refined behaviour as the instances of the superclass. We define a class refinement relation and relate the notion of subclassing to this relation. When a class C' is constructed by subclassing from C and class refinement holds between them, then it is guaranteed that any behaviour expected from C will necessarily be delivered by C'.

Subclassing requires that the parameters of a method be the same in the subclass and in the superclass or, at the most, are subject to contravariance and covariance rules as described in [1, 7, 8]. However, sometimes a change in user requirements causes interface changes of classes

*Turku Centre for Computer Science, Lemminkäisenkatu 14A, Turku 20520, Finland; Anna.Mikhajlova@aton.abo.fi

†Dept. of Computer Science, Åbo Akademi University, Lemminkäisenkatu 14A, Turku 20520, Finland; Emil.Sekerinski@aton.abo.fi

designed as refinements of other classes. We formalize the interface refinement relation as a generalization of class refinement and present guidelines for refinement of clients of the classes involved in this relation. To demonstrate class refinement and interface refinement we specify an example program with client and supplier classes.

Class refinement as defined here is based on data refinement [12, 11, 4, 15, 10]. The definition generalizes that of Sekerinski [18] by allowing contravariance and covariance in the method parameters, and by considering constructor methods. Class refinement has also been studied in various extensions of the Z specification languages, e.g. [13], but only between class specifications and not implementations. Other approaches on "behavioural subtyping" of classes [2, 14] also make a distinction between the specification of a class and its implementation. By having specification constructs as part of the (extended) programming language, this distinction becomes unnecessary.

Naumann [17] defines the semantics of a simple Oberon-like programming language, with similar specification constructs as here, also based on predicate transformers. Sekerinski [18, 19] defines a rich object-oriented programming and specification notation by using a type system with subtyping and type parameters and also using predicate transformers. In both approaches, subtyping is based on extensions of record types. Here we use sum types instead, as suggested by Back and Butler in [5]. One motivation for moving to sum types is to avoid the complications in the typing and the logic when reasoning about record types: the simple typed lambda calculus as the formal basis is sufficient for our purposes. Another advantage of moving to sum types is that we can directly express whether an object is of exactly a certain type or of one of its subtypes (in the record approach, a type contains all the values of its subtypes). Using summations also allows us to model contravariance and covariance on method parameters in a simple way. Finally, to allow objects of a subclass to have different (private) attributes from those of the superclass, hiding by existential types was used in [18, 19]. It turned out that this leads to complications when reasoning about method calls, which are not present when using the model of sum types. In the latter, objects of a subclass can always have different attributes from those of the superclass.

2 Refinement Calculus Basics

A *predicate* over a set of states Σ is a boolean function $p : \Sigma \to Bool$ which assigns a truth value to each state. The set of predicates on Σ is denoted $\mathcal{P}\Sigma$. A *relation* from Σ to Γ is a function $P : \Sigma \to \mathcal{P}\Gamma$ that maps each state σ to a predicate on Γ. We write $\Sigma \leftrightarrow \Gamma \;\hat{=}\; \Sigma \to \mathcal{P}\Gamma$ to denote a set of all relations from Σ to Γ. This view of relations is isomorphic to viewing them as predicates on the cartesian space $\Sigma \times \Gamma$. The *identity relation* and the *composition* of relations are defined as follows:

$$
\begin{aligned}
Id\ x\ y \;&\hat{=}\; x = y \\
(P; Q)\ x\ z \;&\hat{=}\; (\exists y \bullet P\ x\ y \wedge Q\ y\ z).
\end{aligned}
$$

A *predicate transformer* is a function $S : \mathcal{P}\Gamma \to \mathcal{P}\Sigma$ from predicates to predicates. We write $\Sigma \mapsto \Gamma \;\hat{=}\; \mathcal{P}\Gamma \to \mathcal{P}\Sigma$ to denote a set of all predicate transformers from Σ to Γ. Program statements in the refinement calculus are identified with weakest-precondition monotonic predicate transformers that map a postcondition $q : \mathcal{P}\Gamma$ to the weakest precondition $p : \mathcal{P}\Sigma$ such that the program is guaranteed to terminate in a final state satisfying q whenever the initial state satisfies p.

The *refinement ordering* on predicate transformers models the notion of total-correctness preserving program refinement. For programs S and T, the refinement $S \leq T$ holds if and only if T satisfies any specification satisfied by S. Predicate transformers form a complete lattice under the refinement ordering. The bottom element is the predicate transformer **abort** which is never guaranteed to terminate, and the top element is the predicate transformer **magic** which is always guaranteed to establish any postcondition.

Sequential composition of program statements is modeled by functional composition of predicate transformers. The program statement $\mathbf{skip}_\Sigma$ is modeled by the identity predicate transformer on $\mathcal{P}\Sigma$.

Given a relation $P : \Sigma \leftrightarrow \Gamma$, the *angelic update statement* $\{P\} : \Sigma \mapsto \Gamma$ when started in a state σ angelically chooses a new state γ such that $P\ \sigma\ \gamma$ holds, while the *demonic update statement* $[P] : \Sigma \mapsto \Gamma$ demonically chooses a new state γ such that $P\ \sigma\ \gamma$ holds. If no such state exists, then $\{P\}$ aborts, whereas $[P]$ behaves as **magic**, i.e. can establish any postcondition.

Ordinary program constructs may be modeled using the basic predicate transformers and operators presented above. For example, in a state space with two components $(x : T, y : S)$ an assignment statement may be modeled by the demonic update:

$$x := e \quad \widehat{=} \quad [R], \text{ where } R\ (x, y)(x', y') \ = \ (x' = e) \wedge (y' = y)$$

Our specification language includes the *specification statement* $x := x' \bullet b$, where b is a boolean expression relating x and x'. The program variable x is assigned a value x' satisfying b. This statement also corresponds to the demonic update. We also have an *assertion*, written $\{p\}$, where p is a predicate stating a condition on program variables. This assertion behaves as **skip** if p is satisfied and as **abort** otherwise. Finally, the language supports *local variables*. The construct $|[\ \mathbf{var}\ z \bullet S\ ]|$ states that the program variable z is local to S:

$$|[\ \mathbf{var}\ z \bullet S\ ]| \quad \widehat{=} \quad [Enter_z]; S; [Exit_z], \text{ where}$$

$$\begin{array}{lcl} Enter_z(x, y)(x', y', z') & \widehat{=} & (x' = x) \wedge (y' = y) \text{ and} \\ Exit_z(x, y, z)(x', y') & \widehat{=} & (x' = x) \wedge (y' = y) \end{array}$$

The semantics of other ordinary program constructs, like multiple assignments, **if**-statements and **do**-loops is given for example in [3, 4, 15].

Refinement Calculus provides laws for transforming more abstract program structures into more concrete ones based on the notion of refinement of predicate transformers presented above. A large collection of algorithmic and data refinement laws is given in [15] for example.

2.1 Sum Types and Operators

In our specification language we widely employ sum types for modeling subtyping polymorphism and dynamic binding. The sum or disjoint union of two types Σ and Γ is written $\Sigma + \Gamma$. Σ and Γ are called the base types of the sum type.

We define the subtype relation as follows. Σ is a subtype of Σ', written $\Sigma <: \Sigma'$, if $\Sigma = \Sigma'$ or $\Sigma <: \Gamma$ or $\Sigma <: \Gamma'$, where $\Gamma + \Gamma' = \Sigma'$. For example, $\Gamma <: \Gamma + \Gamma'$, and, or course, $\Gamma + \Gamma' <: \Gamma + \Gamma'$. This relation is reflexive, transitive and antisymmetric.

Associated with the sum types are the injection functions which map elements of the subtypes to elements of the supertype summation:

$$\iota_\Sigma : \Sigma \to \Sigma + \Gamma \text{ and } \iota_\Gamma : \Gamma \to \Sigma + \Gamma$$

and projection relations which relate elements of the summation with elements of their subtypes:

$$\pi_\Sigma : \Sigma + \Gamma \leftrightarrow \Sigma \text{ and } \pi_\Gamma : \Sigma + \Gamma \leftrightarrow \Gamma$$

If Σ is a subtype of Σ', we can always construct the appropriate injection $\iota_\Sigma : \Sigma \to \Sigma'$ and projection $\pi_\Sigma : \Sigma' \leftrightarrow \Sigma$.

A summation operator combines statements by forming the disjoint union of their state spaces. This operator is defined in [5] by extension from the summation of types. For $S_1 : \Sigma_1 \mapsto \Gamma_1$ and $S_2 : \Sigma_2 \mapsto \Gamma_2$, the summation $S_1 + S_2 : \Sigma_1 + \Sigma_1 \ \mapsto \ \Gamma_1 + \Gamma_2$ is a predicate transformer such that the effect of executing it in some initial state σ depends on the base type of Σ. If $\sigma : \Sigma_1$ then S_1 is executed, while if it is of type Σ_2, then S_2 is executed.

The summation operator was shown to specify a number of nice properties. The one of interest to us is that it preserves refinement, allowing us to refine elements of the summation separately:

$$S_1 \leq S_1' \ \wedge \ S_2 \leq S_2' \ \Rightarrow \ (S_1 + S_2) \leq (S_1' + S_2').$$

2.2 Product Types and Operators

The cartesian product of two types Σ and Γ is written $\Sigma \times \Gamma$. The product operator combines predicate transformers by forming the cartesian product of their state spaces. For $S_1 : \Sigma_1 \mapsto \Gamma_1$ and $S_2 : \Sigma_2 \mapsto \Gamma_2$, the product $S_1 \times S_2 : \Sigma_1 \times \Sigma_1 \mapsto \Gamma_1 \times \Gamma_2$ is a predicate transformer whose execution has the same effect as simultaneous execution of S_1 and S_2.

In addition to many other useful properties, the product operator preserves refinement:

$$S_1 \leq S_1' \wedge S_2 \leq S_2' \Rightarrow (S_1 \times S_2) \leq (S_1' \times S_2').$$

For $S : \Sigma \mapsto \Sigma$ we define lifting to a product predicate transformer of type $\Sigma \times \Gamma \mapsto \Sigma \times \Gamma$ as $S \times \mathbf{skip}_\Gamma$. When lifting is obvious from the context, we will simply write S instead of $S \times \mathbf{skip}_\Gamma$.

3 Specifying Objects and Classes

Object-oriented systems are characterized by *objects* which group together data and the operations for manipulating that data. The operations, called *methods*, can be invoked only by sending *messages* to the object. The complete set of messages that the object understands is characterized by the *interface* of the object. The interface represents the signatures of object methods i.e. the name and the types of input and output parameters. As opposed to the interface, the *object type* is the type of object *attributes*. We consider all attributes as private or hidden, and all methods as public or visible to clients to the object. Accordingly, two objects with the same public part, i.e. the same interface, can differ in their private part, i.e. object types.

We focus on class-based object-oriented languages which form the mainstream of object-oriented programming. Accordingly, we take a view that objects are instantiated by classes. A class is a pattern used to describe objects with identical behaviour through specifying their interface. Specifically, a class describes what attributes each object will have, the specification for each method and the way the objects are created. We declare a class as follows:

$$
\begin{array}{l}
C \;=\; \mathbf{class} \\
\quad attr_1 : \Sigma_1; \ldots; attr_m : \Sigma_m; \\
\quad C\ (p : \Psi) : \Gamma \;=\; K; \\
\quad Meth_1\ (g_1 : \Gamma_1) : \Delta_1 \;=\; M_1; \\
\quad \ldots \\
\quad Meth_n\ (g_n : \Gamma_n) : \Delta_n \;=\; M_n; \\
\mathbf{end};
\end{array}
$$

Class attributes $attr_1, \ldots, attr_m$ have the corresponding types Σ_1 through Σ_m. The type of all attributes is then $\Sigma = \Sigma_1 \times \ldots \times \Sigma_m$[1]. A *class constructor* is used to instantiate objects and is distinguished by the same name as the class, in our case $C\ (p : \Psi) : \Gamma$. Due to the fact that the constructor concerns object creation rather than object functionality, it is associated with the class rather than with the specified interface. We take a view that the class constructor is not part of the interface specified by the class. Methods $Meth_1$ through $Meth_n$ operate on attributes and realize the object functionality. A signature of every method is part of the specified interface.

The object type specified by a class can always be extracted from a class and we don't need to declare it explicitly. We use $\tau(C)$ to denote the type of objects generated by a class C. In fact, the type of the constructor output parameter which we called Γ is $\tau(C)$. Naturally, $\tau(C)$ is just another name for the type of attributes of C.

Being declared as such, the class C is modeled by a tuple $(K, M_1, \ldots, M_n)$, where the statement K of type $\Psi \mapsto \tau(C)$ specifies a body of the constructor, and the statements M_1 through M_n specify methods $Meth_1, \ldots, Meth_n$.

The constructor takes an input parameter of type Ψ used for initialization of some attributes, declares a new class instance of type $\tau(C)$, initializes its attributes, and returns this new instance.

[1] We must place a non-recursiveness restriction on Σ so that none of Σ_i is equal to Σ. This restriction allows us to stay within the simple-typed lambda calculus.

$$
\begin{array}{l}
Account \ = \ \textbf{class} \\
\quad owner : Name; \ balance : Currency; \ trans : \ \textbf{seq of} \ Transaction; \\[4pt]
\quad Account \ (name : Name, initSum : Currency) : \tau(Account) \ = \\
\qquad |[\ \textbf{var} \ a : \tau(Account) \ \bullet \ a.owner := name; a.balance := initSum; a.trans := \langle\rangle; \ \textbf{return} \ a \ ]|; \\[4pt]
\quad Deposit \ (sum : Currency, from : Name, when : Date) \ = \\
\qquad \{sum > 0\}; \\
\qquad |[\ \textbf{var} \ \ t : Transaction \ \bullet \ t.from := from; t.to := owner; t.amount := sum; \\
\qquad t.date := when; trans := trans^\frown\langle t\rangle; balance := balance + sum \ ]|; \\[4pt]
\quad Withdraw \ (sum : Currency, to : Name, when : Date) \ = \\
\qquad \{sum > 0 \ \wedge \ sum \leq balance\}; \\
\qquad |[\ \textbf{var} \ \ t : Transaction \ \bullet \ t.from := owner; t.to := to; t.amount := sum * (-1); \\
\qquad t.date := when; trans := trans^\frown\langle t\rangle; balance := balance - sum \ ]|; \\
\textbf{end};
\end{array}
$$

Figure 1: Specification of a bank account based on transactions

Introducing a new class instance is modeled by the corresponding class constructor:

$$|[\ \textbf{var} \ c : C(p) \ \bullet \ S \ ]| \ \widehat{=} \ K; [Enter_c]; c := res; [Exit_{res}]; S; [Exit_c].$$

Every method M_i is, in general, of type $\Sigma \times \Gamma_i \ \mapsto \ \Sigma \times \Delta_i$, where Σ is the type of class attributes, Γ_i and Δ_i are the types of input and output parameters respectively. A method may be parameterless with both Γ_i and Δ_i the unit type (), have only input or only output parameters. When a method has output parameters, we expect the statement M_i to include at least one **return** clause. Implicitly, this corresponds to declaring a variable $res : \Delta_i$ in M_i standing for a result, and specifying an assignment $res := e$ as **return** e.

A call to a method $Meth_i \ (g_i : \Gamma_i) : \Delta_i \ = \ M_i$ is modeled as

$$d := c.Meth_i \ (g) \ \widehat{=} \ [Enter_g]; M_i; d := res; [Exit_{res}],$$

where M_i includes **return** res.

As an example of class specification consider a class of bank accounts. An account should have an owner and it should be possible to deposit and withdraw money in the currency of choice. We associate with every account a sequence of transactions, where every transaction has a sender, a receiver, an amount of money being transferred, and a date. A corresponding record type representing transactions is defined as follows:

$$
\begin{array}{l}
\textbf{type} \ Transaction \ = \ \textbf{record} \\
\quad from : Name; \ to : Name; \ amount : Currency; \ date : Date \\
\textbf{end};
\end{array}
$$

Here *Name*, *Currency* and *Date* are simple types. *Date* is a type of six digit arrays for representing a day, a month, and a year, for example as '251296' for December 25, 1996. We present the specification of a class *Account* in Fig. 1. Obviously, this specification only demonstrates the most general behaviour of bank accounts. We would like to *subclass* from *Account* more concrete account classes. Let us consider specification of subclasses more closely.

3.1 Subclassing

Subclassing is a mechanism for constructing new classes from existing ones by *inheriting* some or all of their attributes and methods, *overriding* some attributes and methods and adding new methods. We limit our consideration of class construction to inheritance and overriding. Accordingly, we describe a subclass as follows:

$$
\begin{array}{l}
C' \ = \ \textbf{subclass of } C \\
\quad attr'_1 : \Sigma'_1; \ldots ; \ attr'_p : \Sigma'_p; \\
\quad C' \ (p : \Psi) : \tau(C') \ = K'; \\
\quad Meth_1 \ (g_1 : \Gamma_1) : \Delta_1 \ = M'_1; \\
\quad \ldots \\
\quad Meth_k \ (g_k : \Gamma_k) : \Delta_k \ = M'_k; \\
\textbf{end};
\end{array}
$$

Class attributes $attr'_1, \ldots, attr'_p$ have the corresponding types Σ'_1 through Σ'_p. Some of these attributes are inherited from the superclass C, others override some attributes of C, and some are new. C' has its own class constructor without inheriting the one associated with the superclass. The methods $Meth_1, \ldots, Meth_k$ override the corresponding methods of C and the methods $Meth_{k+1}, \ldots, Meth_n$ are inherited from the superclass C.

We view subclassing as a syntactic relation on classes, since subclasses are distinguished by an appropriate declaration. Syntactic subclassing implies conformance of interfaces, in the sense that a subclass specifies an interface conforming to the one specified by its superclass. In the simple case the interface specified by a subclass is the same as that of the superclass. In the next section we explain how this requirement can be relaxed.

3.2 Modeling Subtyping Polymorphism

To model subtyping polymorphism, we allow object types to be sum types. The idea is to group together an object type of a certain class and object types of all its subclasses to form a polymorphic object type. A variable of such a sum type can be instantiated to any base type of the summation, in other words, to any object instantiated by a class whose object type is the base type of the summation. We will call the object types of only one class *ground* and summations of object types *polymorphic*. Since a ground object type uniquely identifies the class of objects, we can always tell whether a certain object is an instance of a certain class.

A sum of object types, denoted by $\tau(C)^+$ is defined to be such that its base types are $\tau(C)$ and all the object types of subclasses of C. For example, if D is a subclass of C with the object type $\tau(D)$, then $\tau(C)^+ \ = \ \tau(C) \ + \ \tau(D)$. Naturally, we have that

$$\tau(C) <: \tau(C)^+ \text{ and } \tau(D) <: \tau(C)^+.$$

A variable $c : \tau(C)^+$ can be instantiated by either C or D. The *subsumption* property holds of c, namely, if $c : \tau(C)$ and $\tau(C) <: \tau(C)^+$ then $c : \tau(C)^+$. This property is characteristic of subtype relations, it means that an object of type $\tau(C)$ can be viewed as an object of the supertype $\tau(C)^+$. The subsumption property holds in general if all subclasses are known, otherwise it holds for the known ones.

Being equipped with subtyping polymorphism, we can allow overriding methods in a subclass to be generalized on the type of input parameters or to be specialized on the type of output parameters. In the first case this type redefinition is *contravariant* and in the second *covariant*[2]. When one interface is the same as the other except that it can redefine contravariantly input parameter types and covariantly output parameter types, this interface conforms to the original one.

4 Class Refinement

When a subclass overrides some methods of its superclass, there are no guarantees that its instances will deliver the same or refined behaviour as the instances of the superclass. Unrestricted method overriding in a subclass can lead to an arbitrary behaviour of its instances. When used in a

[2]For a more extensive explanation of covariance and contravariance see, e.g. [1].

superclass context, such subclass instances can invalidate their clients. For example, the *Deposit* method of *Account* can be overridden so that the money is, in fact, withdrawn from the account instead of being deposited.

Therefore, we would like to ensure that whenever C' is subclassed from C, any behaviour expected from C will necessarily be delivered by C'. For this purpose, we introduce the notion of class refinement between C and C'.

Consider two classes $C = (K, M_1, ..., M_n)$ and $C' = (K', M_1', ..., M_n')$ with attributes Σ and Σ' respectively, such that $K : \Psi \mapsto \Sigma$ and $K' : \Psi' \mapsto \Sigma'$ are the class constructors and all $M_i : \Sigma \times \Gamma_i \mapsto \Sigma \times \Delta_i$ and $M_i' : \Sigma' \times \Gamma_i' \mapsto \Sigma' \times \Delta_i'$ are the corresponding methods. The input parameters of the constructors are either the same or contravariant, such that $\Psi <: \Psi'$. The input parameters of the methods are either the same or contravariant, such that $\Gamma_i <: \Gamma_i'$, and the output parameters are either the same or covariant, $\Delta_i' <: \Delta_i$.

We define the refinement of class constructors K and K' with respect to abstraction relations R and π_Ψ as follows:

$$K \leq_{R,\pi_\Psi} K' \;\; \widehat{=} \;\; \{\pi_\Psi\};\; K \;\; \leq \;\; K';\; \{R\} \tag{1}$$

where $R : \Sigma' \leftrightarrow \Sigma$ is an abstraction relation coercing attributes of C' to the attributes of C, and $\pi_\Psi : \Psi' \leftrightarrow \Psi$ is a relation projecting Ψ from Ψ'.

The refinement of all corresponding methods M_i and M_i' with respect to abstraction relations R, π_{Γ_i} $\iota_{\Delta_i'}$ is defined as

$$M_i \leq_{R,\pi_{\Gamma_i},\iota_{\Delta_i'}} M_i' \;\; \widehat{=} \;\; \{R \times \pi_{\Gamma_i}\};\; M_i \;\; \leq \;\; M_i';\; \{R \times \iota_{\Delta_i'}\} \tag{2}$$

where $R : \Sigma' \leftrightarrow \Sigma$ is as before, $\pi_{\Gamma_i} : \Gamma_i' \leftrightarrow \Gamma_i$ is a projection relation and $\iota_{\Delta_i'} : \Delta_i' \leftrightarrow \Delta_i$ is an injection relation. Obviously, when $\Gamma_i = \Gamma_i'$, the projection relation π_{Γ_i} is taken to be the identity relation Id. The same holds when $\Delta_i = \Delta_i'$, namely, $\iota_{\Delta_i'} = Id$.

Now the class refinement relation $C \sqsubseteq C'$ holds between classes C and C' if:

- The constructor of C' refines the constructor of C as defined in (1).
- Every method M_i' of C' refines the corresponding method M_i of C as defined in (2).

Based on method refinement, the class refinement relation shares the properties of statement refinement and is, thus, reflexive and transitive.

Declaring one class as a subclass of another raises the proof obligation that the class refinement relation holds between these classes. This is in a way a semantic constraint that we impose on subclassing to ensure that behaviour of subclasses conforms to the behaviour of their superclasses and, respectively, that the subclasses can be used in the superclass context.

5 Interface Refinement

Subclassing requires that the parameters of a method be the same in the subclass and in the superclass or, at the most, subject to contravariance and covariance rules. However, sometimes the change in user requirements causes the interface change of classes designed as refinements of other classes.

When the new interface is similar to the old interface, we can identify abstraction relations coercing new parameters to the old ones. Let us define the interface refinement relation between classes with respect to these relations.

Consider two classes $C = (K, M_1, \ldots, M_n)$ and $C' = (K', M_1', \ldots, M_n')$ with attributes Σ and Σ' respectively, such that $K : \Psi \mapsto \Sigma$ and $K' : \Psi' \mapsto \Sigma'$ are the class constructors and all $M_i : \Sigma \times \Gamma_i \mapsto \Sigma \times \Delta_i$ and $M_i' : \Sigma' \times \Gamma_i' \mapsto \Sigma' \times \Delta_i'$ are the corresponding methods.

Let $R : \Sigma' \leftrightarrow \Sigma$ be an abstraction relation coercing attributes of C' to those of C, and $I_O : \Psi' \leftrightarrow \Psi$ an abstraction relation coercing the corresponding input parameter types. We define the refinement of class constructors K and K' through R and I_0 as follows:

$$K \leq_{R,I_0} K' \;\; \widehat{=} \;\; \{I_0\};\; K \;\; \leq \;\; K';\; \{R\} \tag{3}$$

$NDAccount \;=\;$ **class**
$\quad owner : Name;\; balance : Currency;\; trans :$ **seq of** $NewTran;$

$\quad NDAccount\,(name : Name, initSum : Currency) : \tau(NDAccount) \;=$
$\qquad |[$ **var** $a : \tau(NDAccount) \;\bullet\; a.owner := name; a.balance := initSum; a.trans := \langle\rangle;$ **return** $a\;]|;$

$\quad Deposit\,(sum : Currency, from : Name, when : NewDate) \;=$
$\qquad \{sum > 0\};$
$\qquad |[$ **var** $\; t : NewTran \;\bullet\; t.from := from; t.to := owner; t.amount := sum;$
$\qquad t.date := when; trans := trans\,^\frown\langle t\rangle; balance := balance + sum\;]|;$
end;

Figure 2: Specification of an account based on NewTran

Obviously, (3) is a generalization of (1) with $I_0 = Id$ when $\Psi = \Psi'$ or with $I_0 = \pi_\Psi$ when the input types are contravariant.

Let $R : \Sigma' \leftrightarrow \Sigma$ be as before and $I_i : \Gamma_i' \leftrightarrow \Gamma_i$ and $O_i : \Delta_i' \leftrightarrow \Delta_i$ be abstraction relations coercing the corresponding input and output parameter types. We define the refinement of corresponding methods M_i and M_i' through R, I_i and O_i as follows:

$$M_i \leq_{R,I_i,O_i} M_i' \quad \widehat{=} \quad \{R \times I_i\};\; M_i \quad \leq \quad M_i';\; \{R \times O_i\} \tag{4}$$

Obviously, (4) is a generalization of (2) with $I_i = \pi_{\Gamma_i}$ when the inputs are contravariant, $\Gamma_i <: \Gamma_i'$, and with $O_i = \iota_{\Delta_i'}$ when the outputs are covariant, $\Delta_i' <: \Delta_i$.

Now the interface refinement relation $C \sqsubseteq_{I,O} C'$ with respect to parameter abstraction relations $I = (I_0, I_1, \ldots, I_n)$ and $O = (O_1, \ldots, O_n)$ holds if:

- The constructor of C' refines the constructor of C as defined in (3).
- Every method M_i' of C' refines the corresponding method M_i of C as defined in (4).

Being defined as such, interface refinement of classes is a generalization of class refinement. When every I_i and O_i is the identity relation or the projection and injection relations respectively, interface refinement is specialized to class refinement.

As an example of interface refinement consider our previous specification of transactions and accounts. Suppose that facing the start of the new century, we'd like to change the type of dates so that it is possible to specify a four-digit year:

type $NewDate \;=\;$ **array** $[1..8]$ **of** $Digit;$

Accordingly, we have to define a new transaction record type

type $NewTran \;=\;$ **record**
$\quad from : Name;\; to : Name;\; amount : Currency;\; date : NewDate$
end;

and construct a new class of accounts as shown in Fig.2. We omit the specification of the *Withdraw* method, which is similar to that of *Account* with a local variable of type *NewTran* rather than *Transaction*. It can be shown that $Account \;\sqsubseteq_{I,O}\; NDAccount$, where $I = (Id \times Id, Id \times Id \times D, Id \times Id \times D)$ and $O = (Id, Id)$. The abstraction relation D is defined so that for constants '1' and '9' of type *Digit* and for any $d : Date$ and $d' : NewDate$

$$D(d')(d) \quad \widehat{=} \quad (d'[1..4] = d[1..4]) \wedge (d'[5..6] = \text{`1'`9'}) \wedge (d'[7..8] = d[5..6]).$$

Now let us consider how clients can benefit from the interface refinement of their supplier classes without being changed in any way. We call this kind of client refinement *implicit*, and it takes place when it is impractical or impossible to redefine clients of *OldClass*, but it is however desirable that they work with *NewClass* which is the interface refinement of *OldClass*. We can implicitly refine clients by employing a so-called *forwarding scheme*. The idea is to introduce a subclass of *OldClass*, *NewWrapper* which aggregates an instance of *NewClass* and forwards

$$
\begin{array}{l}
AccountWrapper \ = \ \textbf{subclass of } Account \\
\quad impl : \tau(NDAccount); \\[4pt]
\quad AccountWrapper \ (name : Name, initSum : Currency) : \tau(AccountWrapper) \ = \\
\quad\quad |[\ \textbf{var } a : \tau(AccountWrapper), \ a.impl : NDAccount(name, initSum) \ \bullet \ \textbf{return } a \ ]|; \\[4pt]
\quad Deposit \ (sum : Currency, from : Name, when : Date) \ = \\
\quad\quad \{sum > 0\}; \ impl.Deposit(sum, from, ToNewDate(when)); \\[4pt]
\quad Withdraw \ (sum : Currency, to : Name, when : Date) \ = \\
\quad\quad \{sum > 0 \ \wedge \ sum \leq balance\}; \ impl.Withdraw(sum, to, ToNewDate(when)); \\
\textbf{end};
\end{array}
$$

Figure 3: Specification of a wrapper class for implicit interface refinement

OldClass method calls to *NewClass* through this instance. This has also been identified as a reoccuring design pattern by Gamma et al. [9].

Clients of *OldClass* can work with *NewWrapper* which is a subclass of *OldClass* but have all the benefits of working with *NewClass* if:

$$OldClass \ \sqsubseteq \ NewWrapper \text{ and } NewWrapper \ \sqsubseteq_{I,O} \ NewClass$$

Consider again our example. Clients of *Account* want to use *NDAccount* but cannot do so since the latter specifies an interface different from that specified by *Account*. We can employ the forwarding scheme by introducing a new class *AccountWrapper* in Fig. 3 which aggregates an instance of *NDAccount* and forwards *Account* method calls to *NDAccount* via this instance. Clients of *Account* can in this case be implicitly refined to work with *NDAccount* via *AccountWrapper*.

6 Conclusions

We have defined the class refinement relation and the interface refinement relation which allow a developer to construct extensible object-oriented programs from specifications and assure reliability of the final program. Proving that class refinement holds between a subclass and its superclass ensures that any behaviour expected from a superclass will be delivered by its subclass.

The user requirements captured in the initial specification are preserved and refined at every stage of development using the class refinement relation. If a change in the user requirements causes the interface change of some classes, the presented guidelines for implicit interface refinement of clients can be used to refine the specification of the whole program. We feel that there is a strong connection between parameter abstraction relations with respect to which interface refinement is defined and explicit refinement of clients of the refined classes. Investigating how clients can be explicitly refined based on the parameter abstraction relations for the supplier classes is the topic of current research.

Acknowledgments

We would like to thank Ralph Back for a lot of useful comments.

References

[1] Martín Abadi and Luca Cardelli. A Theory of Objects. Springer-Verlag, 1996.

[2] Pierre America. Designing an object-oriented programming language with behavioral subtyping. In J.W de Bakker, W. P. de Roever, and G. Rozenberg, editors, *Foundations of Object-Oriented Languages, REX School/Workshop, Noordwijkerhout, The*

Netherlands, May/June 1990, volume 489 of Lecture Notes in Computer Science, pages 60-90. Springer-Verlag, New York, N.Y., 1991.

[3] R. J. R. Back. Correctness Preserving Program Refinements: Proof Theory and Applications. *vol. 131 of Mathematical Center Tracts.* Amsterdam: Mathematical Center, 1980.

[4] R. J. R. Back. Changing data representation in the refinement calculus. In *21st Hawaii International Conference on System Sciences*, 1989.

[5] R. J. R. Back and M. J. Butler. Exploring summation and product operators in the refinement calculus. In B. Möller, editor, *Mathematics of Program Construction, 1995*, volume LNCS 947. Springer–Verlag, 1995.

[6] K. B. Bruce, L. Cardelli, G. Castagna, The Hopkins Object Group, G.T. Leavens and B.C. Pierce. On binary methods. *Theory and Practice of Object Systems* 1(3):221–242, 1996.

[7] L. Cardelli and P. Wegner. On understanding types, data abstraction, and polymorphism. *Computing Surveys,* 17(4):471–522, 1985.

[8] G. Castagna. Covariance and contravariance: conflict without a cause. *ACM Transactions on Programming Languages and Systems,* 17(3):431–447, March 1995.

[9] E. Gamma, R. Helm, R. Johnson and J. Vlissides. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, Professional Computing Series, 1995.

[10] P.H.B. Gardiner and C.C. Morgan. Data refinement of predicate transformers. *Theoretical Computer science*, 87:143-162, 1991.

[11] Jifeng He, C.A.R.Hoare and J.W.Sanders. Data Refinement Refined. In *European Symposium on Programming*, Lecture Notes in Computer Science 213, Springer–Verlag, 1986.

[12] C.A.R. Hoare. Proofs of correctness of data representation. *Acta informatica,* 1(4), 1972.

[13] K. Lano and H. Haughton. Reasoning and Refinement in Object-Oriented Specification Languages. *European Conference on Object-Oriented Programming '92,* Lecture Notes in Computer Science 615, Springer–Verlag, 1992.

[14] Barbara Liskov and Jeannette M. Wing. Specifications and their use in defining subtypes. *OOPSLA '93 Proceedings, ACM SIGPLAN Notices,* 28(10):16-28, October 1993.

[15] Carroll C.Morgan. Programming from Specifications. Prentice-Hall, 1990.

[16] J. M. Morris. A Theoretical Basis for Stepwise Refinement and the Programming Calculus. In *Science of Computer Programming,* 9, 287–306, 1987.

[17] D.A.Naumann. Predicate Transformer Semantics of an Oberon-Like Language. In Ernst-R. Olderog, editor, *Programming Concepts, Methods and Calculi,* International Federation for Information Processing, 460-480, 1994.

[18] Emil Sekerinski. Verfeinerung in der Objektorientierten Programmkonstruktion. *Ph.D. Thesis.* Universität Karlsruhe, 1994.

[19] Emil Sekerinski. A Type-Theoretic Basis for an Object-Oriented Refinement Calculus. In S.J. Goldsack and S.J.H. Kent, editors, *Formal Methods and Object Technology.* Springer-Verlag, 1996.