Patterns and Simulation Relations for Behavioural Subtyping

Heike Wehrheim

Universität Oldenburg
Fachbereich Informatik
Postfach 2503, D–26111 Oldenburg, Germany
wehrheim@informatik.uni-oldenburg.de

Category 2

Abstract. Subtyping relations for object-oriented formalisms describe relationships between super- and subclasses which satisfy the substitutability requirement imposed on types and their subtypes. *Behavioural* subtyping is concerned with subtypes for *active* classes with an explicit dynamic behaviour, specifiable for instance by object-oriented formal methods combining state-based with behavioural formalisms. Such specification languages often have a formal semantics that composes behavioural semantics for both the behaviour *and* the state-based part.
In this paper we develop syntactic *patterns* for the state-based part of a subclass which guarantee the subclass to be a behavioural subtype of its superclass *by construction.* Our results are similar to the ones linking data refinement in state-based methods with failure-divergence refinement in CSP. In contrast to classical data refinement, subtyping (and the simulation relations used for showing the soundness of the patterns) have to cope with additional new operations in the subclass.

1 Introduction

Inheritance is one of the key features for the success of object-oriented analysis and design methods as well as programming languages. Inheritance is used to reuse already existing class descriptions and extend them with additional attributes and methods, thus creating a new subclass from an existing superclass. Subtyping for object-oriented formalisms [Ame91,DL96,LW94] studies relationships between super- and subclasses with the aim of clarifying when a subclass can be said to be *type substitutable* for its superclass. Type substitutability means that "a subtype instance can be used wherever a supertype instance is expected" [WZ88]. Subtyping is also studied for *active* classes with a behavioural semantics. Active classes have an explicit specification of the dynamic behaviour which for instance describes synchronisation constraints of the class or certain protocols which have to be obeyed. In this paper, we use the term behavioural subtyping to refer to subtyping for this type of classes. Subtyping is necessary for ensuring a safe usage of type polymorphism in languages with dynamic referencing. Knowledge about subtypes can furthermore be useful in an incremental design: certain properties of superclasses are known to be inherited to their subtypes [Weh00]. Moreover, substitutivity opens another broad application area for subtyping relations in object-orientation: in component-based software design the choice of components from libraries is driven by the demand that the delivered component either has to exactly match the requested one or has to be substitutable for it. For instance Sun's new JINI architecture [Wal99] explicitly builds on a notion of subtype, which fixes the correctness of services: any request for a special interface type may be granted by returning a subtype.

A number of proposals for *behavioural subtyping* relations have been made (among others see [vdAB97,BCP99,BBSDS97,Cus91,HK99,Nie95]), with for instance Petri nets or labelled transition systems as semantic basis. The starting point of this paper are two subtyping relations from [FW00] with proven substitutability properties, one of them also adequate for shared mutable objects. The formal basis for these two relations are transition systems and their (CSP) failures and divergences.

Here we will be concerned with *checking* subtype relationships among classes specified in object-oriented formal methods combining state-based with behavioural aspects. The formal method which we have in mind here (and which we use throughout our examples) is CSP-OZ [Fis97,Fis00], a combination of CSP [Hoa85,Ros97] and Object-Z [Smi00], but the theory we develop will be independent of this particular choice. Two aspects of CSP-OZ are however of importance: first, the fact that the state-based part of CSP-OZ has a *behavioural semantics* (and thus we have to use *behavioural* subtyping relations), and second, that the semantics of the combination is obtained by separately computing the semantics of the Object-Z and CSP part, and afterwards joining them by CSP parallel composition. This second point allows us to look at state-based and behaviour part in isolation when checking for subtypes.

A straightforward approach to find out about subtyping relationships among classes is the following: given two class descriptions, compute their behavioural semantics and compare the classes according to some chosen relation. However, since the semantics of CSP-OZ (and most other such types of integrated formal methods) has to be constructed by hand[1], this approach is in general impracticable. Instead we are interested in developing simple *syntactic patterns*, which allow to avoid the explicit check and instead guarantee subtypes *by construction*. Such patterns have to give construction rules on attributes and operations in subclasses, which, when followed, immediately lead to subtypes. Subtyping patterns even facilitate subtype checks for object-oriented formal methods with tools for computing the semantics since this computation can be rather time-consuming. The use of patterns in object-oriented design is a common technique [GHJV85], and here it also allows a designer with no background in formal methods to construct subtypes.

In this paper we will solely be interested in subtyping patterns for the state-based part (which, due to the compositional semantics, can without change be combined with patterns for the behaviour part). Our investigations are similar to the work about the relationship between data refinement rules in state-based formalisms and failure-divergence refinement [He89,Jos88,WM90,DBBS97]. There it has been studied whether data refinement induces failure-divergence refinement when the state-based formalism is given a behavioural semantics. The basic approach taken here is the same: we assume to have a behavioural semantics for a state-based formalism, have a set of rules on two class specifications and show that they induce a behavioural subtyping relation on their semantics (soundness of rules). Our aim is not to find a *complete* set of rules, but to give simple *syntactic* patterns which allow for an easy construction of a subtype from a given superclass. In contrast to data refinement, subtyping has to deal with additional new methods in the subclass. This also has to be taken into account when we define simulation relations for use in the soundness proofs. The simulation relations require the superclass to simulate the subclass *up to* the new methods, where "up to" is defined in two different ways depending on which subtyping relation should be supported.

We illustrate our approach by a small example of a buffer which is extended by means of inheritance in two different ways yielding two different kinds of subtypes. The construction of the subclasses both follow patterns.

[1] An exception are integrated formal methods with modelcheckers.

The paper is structured as follows: We start with the buffer specification and thereby explain our abstract descriptions of classes and their semantics. The next section gives necessary definitions and explains the two subtyping relations. Section 4 introduces the patterns and shows their correctness. For the correctness proof we define two simulation relations inducing behavioural subtyping.

2 Classes

We start with an example introducing the specification of a simple buffer and two extensions constructed by inheritance. This example will be used to illustrate on the one hand our abstract class descriptions and their semantics, and on the other hand the subtyping relations and patterns. We use CSP-OZ [Fis00,Fis97] syntax in the specifications. Nevertheless, the concrete syntax of specifications is completely unimportant for the results of this paper since we formally work with a more abstract description of classes. The following class specifications will *not* contain any CSP behaviour descriptions since we are only interested in patterns for the state-based part. Thus the classes almost look like Object-Z classes.

The buffer to be specified can hold elements of a basic type *Element* up to some arbitrary number n. Elements may be put into the buffer and retrieved by means of the methods *put* and *get*, and an attribute *alive* is used to indicate whether some specific object of class *Buffer* is still living.

$[Element]$

$Buffer(n : \mathbb{N})$

method $put : [\, el? : Element \,]$
method $get : [\, el! : Element \,]$

$contents : \mathbb{P}\, Element$
$alive : \mathbb{B}$

$\# contents \leq n$

Init
$contents = \varnothing$
$alive = \mathrm{true}$

enable_put
$el? : Element$

$\# contents < n \wedge alive$

effect_put
$\Delta(contents)$
$el? : Element$

$contents' = contents \cup \{el?\}$

enable_get

$\# contents \geq 1 \wedge alive$

effect_get
$\Delta(contents)$
$el! : Element$

$contents' = contents \setminus \{el!\}$

The class is parameterised in the length of the buffer. The first two lines define the *interface* of the class: it has methods *put* and *get*. Next the attributes and their initial values are defined. The application of methods *put* and *get* is described by enable and effect predicates, the enabling predicate specifying a *guard* for the application of the method, the effect predicate the possible results of applying the method. For instance, method *put* is only enabled if the buffer is not yet full and still alive, and the execution of *put* adds a new element to the set *contents*.

Abstractly we represent a class C by a *state*, an *initialisation predicate* and a number of *operations*: $C = \big(State, Init, (Op_j)_{j \in J}\big)$. The components have the following meaning:

- *State* describes the state space of the class, i.e. the possible evaluations of attributes. We let *Attr* stand for the set of attributes of the class.
- *Init* is a predicate over *Attr*, *In* denotes the set of states which satisfy *Init*.
- Op_j is an operation with input $x_j?$ from some set X_j and output $y_j!$ from Y_j. Furthermore for every operation we have
  1. an enabling predicate $\texttt{enable\_}op_j$ over *Attr* and $x_j?$ which acts as an guard to the operation execution and
  2. an effect predicate $\texttt{effect\_}op_j$ over *Attr*, $Attr'$, $x_j?$ and $y_j!$ which describes the postcondition of the operation (where $Attr'$ refers to the new state of attributes).

  The precondition of op_j (possibly different from the enabling condition) is defined as $\texttt{pre\_}op_j := \exists\, Attr', y_j! : \texttt{effect\_}op_j$.

Next we provide the class description with a behavioural semantics in terms of labelled transition systems. The labels are taken from an alphabet Σ which contains all possible names of methods (to be precise, it contains the *events*, i.e. method names plus parameters) and we furthermore assume to have two symbols τ, ν for invisible actions at hand, letting $\Sigma_{\tau\nu}$ denote $\Sigma \cup \{\tau, \nu\}$. The symbol τ plays the usual role of an internal action in process algebra, in particular, the ability of performing an infinite number of τ's from some state on stands for divergence. The role of ν will become clear when we define the subtyping relations.

Definition 1. *A* labelled transition system *(LTS) is a tuple* $T = (Q, \longrightarrow, Q^{in})$ *such that*

- Q *is a set of states,*
- $\longrightarrow \subseteq Q \times \Sigma_{\tau\nu} \times Q$ *is the transition relation and*
- $Q^{in} \subseteq Q$ *is the set of initial states.*

The alphabet *of* T, $\alpha(T)$, *is the set of labels occurring in* T.

The basic idea of the semantics is to view a class as a *process* who is continously executing enabled operations and thus changes its state. Initially it is in one of the states as described by *In*. The symbol $\perp$ is used to indicate one particular *divergent* state: The class diverges if an enabled operation is applied outside its precondition. This distinction between enabling predicates acting as guards and operation preconditions describing the states with a safe application of the operation is a fundamental choice made for the CSP-OZ semantics and allows to describe a blocking as well as a non-blocking view of operations[2].

Definition 2.
The operational semantics of a class C *is defined as* $[\![C]\!] = (Q_C, \longrightarrow_C, Q_C^{in})$, *where*

- $Q_C = State \cup \{\perp\}$,
- $Q_C^{in} = In$ *and*
- $\longrightarrow_C$ *is defined as*
  - $q \xrightarrow{ev}_C q'$ *if* $ev = op_j.i.o$, $i \in X_j$, $o \in Y_j$ *and* $\texttt{enable\_}op_j(q, i)$ *and* $\texttt{effect\_}op_j(q, q', i, o)$,
  - $q \xrightarrow{ev}_C \perp$ *if* $ev = op_j.i.o$, $i \in X_j$, $o \in Y_j$ *and* $\texttt{enable\_}op_j(q, i)$ *but not* $\texttt{pre\_}op_j(q, i)$,
  - $\perp \xrightarrow{\tau}_C \perp$.

[2] Nevertheless, the semantics given here does not completely match the semantics of CSP-OZ since for reasons of simplicity we have not included CSP-OZ' treatment of outputs.

The evaluation of predicates (e.g. $\texttt{enable\_}op(q, i)$) proceeds by substituting attribute names with their concrete values according to the binding in state q and the values of i and o, respectively. In this definition an explicit distinction between operations and events (operation names with parameters) is made. Since this distinction is unimportant for the rest of the paper we will identify events and operations in the following.

The dynamic behaviour of class $\mathit{Buffer}(1)$ derived by Definition 2 is depicted by the transition system in Figure 1.

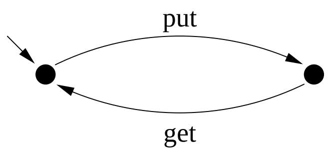

Fig. 1. Dynamic behaviour of superclass $\mathit{Buffer}(1)$

Next we take a look at inheritance. The basic idea of *inheritance* is that in defining a new class it is often very convenient to start with all attributes and methods of an already existing class and to add some more in order to get the new class. Here we present two extensions of the basic class *Buffer*: the first adds a possibility of explicitly deleting the buffer, the second adds a new method checking for emptiness.

$\mathit{FinBuffer}(n : \mathbb{N})$

$\texttt{inherit}$ $\mathit{Buffer}(n)$

$\texttt{method}$ del

$\texttt{enable\_}\mathit{del}$

$\#\mathit{contents} = 0 \wedge \mathit{alive}$

$\texttt{effect\_}\mathit{del}$

$\Delta(\mathit{alive})$

$\mathit{alive} = \mathit{false}$

$\mathit{EmptyCheckBuffer}(n : \mathbb{N})$

$\texttt{inherit}$ $\mathit{Buffer}(n)$

$\texttt{method}$ $\mathit{empty} : [\, b! : \mathbb{B} \,]$

$\texttt{enable\_}\mathit{empty}$

alive

$\texttt{effect\_}\mathit{empty}$

$\Delta()$
$b! : \mathbb{B}$

$b! = (\mathit{contents} = \varnothing)$

Conceptually inheritance is defined as conjunction: given an abstract class $A = \big(\mathit{AState}, \mathit{AInit}, (\mathit{AOp}_j)_{j \in J}\big)$ and specialised class $C = \big(\mathit{CState}, \mathit{CInit}, (\mathit{COp}_i)_{i \in I}\big)$ assuming C is constructed from A by means of inheritance, then we have $J \subseteq I$ and

$$\begin{aligned}
\mathit{CState} &= \mathit{AState} \wedge \mathit{ExtState}, \\
\mathit{CInit} &= \mathit{AInit} \wedge \mathit{ExtInit}, \\
\mathit{COp}_j &= \mathit{AOp}_j \wedge \mathit{ExtOp}_j \qquad \forall\, j \in J, \\
\mathit{COp}_i &= \mathit{ExtOp}_i \qquad \forall\, i \in I \setminus J.
\end{aligned}$$

ExtState (*ExtInit* etc.) is the part which is explicitly written in the subclass and *AState* (etc.) is inherited. The conjunction on the operation part separately treats the enabling and effect predicates.

Using again Definition 2 we obtain the transition systems in Figure 2 and 3 as operational semantics for *FinBuffer*(1) and *EmptyCheckBuffer*(1).

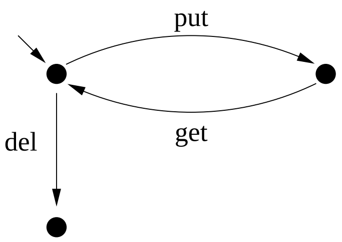

Fig. 2. Dynamic behaviour of subclass *FinBuffer*

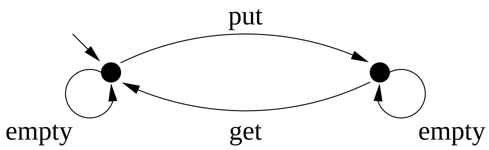

Fig. 3. Dynamic behaviour of subclass *EmptyCheckBuffer*

3 Subtyping

Subtyping on ordinary types is usually demanded to fulfill the requirement of *type substitutability*: subtypes should be usable wherever supertypes are expected without any noticeable differences [WZ88]. When lifting this concept to classes, the first idea was to use the inheritance hierarchy for defining subtyping relations. This however does not fit the requirement of substitutivity anymore: inheritance is first of all concerned with *re-use* of class descriptions and allows fundamental changes to the behaviour of superclasses in a subclass. The requirement of substitutivity suggests that instead some sort of refinement should be used as a subtyping relation. In the context of behavioural subtyping this should be process refinement. However, plain refinement is not appropriate for subtyping as well since refinement does not allow *extension of functionality*, i.e. the addition of new methods in the subclass. Nevertheless, both our subtyping relations are based on refinement, hence we start with its definition.

First we need some auxiliary definitions on transition systems.

Definition 3. *Let* $T = (Q, \longrightarrow, Q^{in})$ *be a labelled transition system,* $q, q' \in Q$, $a_i \in \Sigma_{\tau\surd}$ *and* $\sigma \in \Sigma^*$.

- $q \xrightarrow{a_1 \ldots a_n} q'$ *iff there are states* $q_0, q_1, \ldots, q_n$ *such that* $q_0 \in Q^{in}$, $q_i \xrightarrow{a_{i+1}} q_{i+1}$ *and* $q_n = q'$.

- $q \stackrel{\sigma}{\Longrightarrow} q'$ *iff there is a trace* $t \in \Sigma^*_{\tau\nu}$ *such that* $q \stackrel{t}{\longrightarrow} q'$ *and* $\sigma = t \downarrow \Sigma$ *($\downarrow$ denoting projection).*
- *A state is* stable *if no* τ *transitions are possible:* q *stable iff* $q \stackrel{\tau}{\not\longrightarrow}$.
A state diverges *if an infinite number of* τ*'s are possible from it:* $q\uparrow$ *iff* $\forall\, n \in \mathbb{N} : q \stackrel{\tau^n}{\longrightarrow}$.
The set of enabled actions *of a state* $q \in Q$ *is*

$$next(q) := \{a \in \Sigma \mid \exists\, q' \in Q : q \stackrel{a}{\longrightarrow} q'\},$$

its maximal refusals *are* $refusals(q) := \Sigma \setminus next(q)$.
- *The set of* stable failures *of* T *is*

$$failures(T) := \{(\sigma, X) \in \Sigma^* \times \mathbf{2}^\Sigma \mid \exists\, q_0 \in Q^{in}, q \in Q : \; q_0 \stackrel{\sigma}{\Longrightarrow} q \wedge q \text{ stable } \wedge \\ X \subseteq refusals(q)\}.$$

- *The set of* divergences *of* T *is*

$$divergences(T) := \{\sigma \frown \sigma' \in \Sigma^* \mid \exists\, q_0 \in Q^{in}, q \in Q : q_0 \stackrel{\sigma}{\Longrightarrow} q \wedge q\uparrow\}$$

We have omitted a closure of failures with respect to divergent traces here, thus keeping matters simpler. $Buffer(1)$ for instance has failures (with maximal refusals)

$$(\varepsilon, \{get\}) \quad (put, \{put\}) \quad (put \frown get, \{get\}) \quad \ldots$$

and has an empty set of divergences.

In the process algebra CSP process refinement is used to compare an implementation against a specification with respect to their failures and divergences: the implementation has to be more deterministic than the specification, i.e. has less failures and divergences.

Definition 4. *A labelled transition system* I *is a* failure-divergence refinement *of an lts* S *(*$S \sqsubseteq_{\mathcal{FD}} I$*) iff*

$$failures(I) \subseteq failures(S) \text{ and} \\ divergences(I) \subseteq divergences(S) \; .$$

According to this definition neither *FinBuffer* nor *EmptyCheckBuffer* are refinements of *Buffer* (e.g. *FinBuffer* has a failure (del, Σ) which is not in *Buffer*'s failure set). Nevertheless refinement will be the basis for our two behavioural subtyping relations. The relations additionally have to cope with the fact that the subclass may introduce new methods. This new methods have to be hidden somehow when comparing super- and subclass. Two operators on transition systems are used for this purpose: restriction and concealment.

Definition 5. *Let* $T = (Q, \longrightarrow, Q^{in})$ *be a labelled transition system and* $A \subseteq \Sigma$.

1. *The* restriction of A in T, $T \setminus_r A$, *is defined as* $(Q, \longrightarrow', Q^{in})$ *with*

$$\longrightarrow' = \{(q, a, q') \in \longrightarrow \mid a \notin A\} \; .$$

2. *The* concealment of A in T, $T \setminus_c A$, *is defined as* $(Q, \longrightarrow', Q^{in})$ *with*

$$\longrightarrow' = \{(q, a, q') \in \longrightarrow \mid a \notin A\} \cup \{(q, \nu, q') \mid \exists\, a \in A : (q, a, q') \in \longrightarrow\} \; .$$

Note that concealment renames actions into ν not τ actions. The concealed actions are afterwards not internal to the object but simply invisible. Thus concealment differs from classical hiding in process algebras, which assumes that the hidden actions are under control of the object alone. Hiding may usually introduce new non-stable states, whereas concealment does not. The symbol ν is solely used for this purpose of introducing concealment as an operator that does not create new instabilities.

Superclass and subclass are then compared by applying concealment or restriction to the new methods and checking the results for refinement.

Definition 6. *Let A, C be labelled transition systems, A standing for the super- and C for the subclass, and let $N = \alpha(C) \setminus \alpha(A)$ be the set of new methods.*

- *C is a* weak subtype *of A (denoted $A \sqsubseteq_{wst}^{N} C$) iff $A \sqsubseteq_{\mathcal{FD}} C \setminus_r N$.*
- *C is an* optimal subtype *of A (denoted $A \sqsubseteq_{ost}^{N} C$) iff $A \sqsubseteq_{\mathcal{FD}} C \setminus_c N$.*

Coming back to our three buffers: *FinBuffer* is a weak subtype of *Buffer* and *EmptyCheckBuffer* is a weak as well as an optimal subtype. *FinBuffer* is not an optimal subtype of *Buffer* since $FinBuffer \setminus_c \{del\}$ has a failure (ε, Σ) (renaming *del* to ν) which *Buffer* does not have.

As the names already suggest, optimal subtyping is stronger than weak and the two relations therefore also achieve substitutivity to different degrees. While weak subtypes are substitutable for superclass objects only as long as there is only one client using the object, optimal subtypes are also substitutable when the object is shared among many clients (for details see [FW00]).

In order to check for subtypes we first had to construct the transition systems for the classes and then had to apply the subtyping definition. Next we look at patterns which give us the possibility of obtaining a subtype *by construction*.

4 Subtyping patterns

We present two patterns here, one for each subtyping relation. The patterns are essentially rules on the changes and extensions allowed in the subclass. We again assume $A = (AState, AInit, (AOp_j)_{j \in J})$ to be the super- and, letting $J \subseteq I$, $C = (CState, CInit, (COp_i)_{i \in I})$ to be the subclass.

Patterns:

P1 $\forall j \in J : COp_j = AOp_j$, i.e. the specialised subclass does not change the old methods at all. There are no restrictions on new methods or attributes.

P2

P2.1 $\forall j \in J : COp_j = AOp_j$,

P2.2 $\forall n \in I \setminus J$: `effect_`COp_n is a predicate solely over *Attr*, *ExtAttr* and *ExtAttr'*, i.e. no changes to old attributes are allowed,

P2.3 $\forall n \in I \setminus J$: `enable_`$COp_n \Rightarrow$ `pre_`COp_n.

Pattern P1 is a pattern for a weak subtype, e.g. *FinBuffer* may be constructed from *Buffer* using pattern P1, it leaves all old methods unchanged. Pattern P2 is valid for *EmptyCheckBuffer*: the new method *empty* does not change old attributes and its precondition holds whenever the method is enabled. We however cannot construct *FinBuffer* from *Buffer* using pattern P2 since *FinBuffer*'s effect predicate for operation *del* changes a value of an old attribute (*alive*). Condition P2.3 is essentially used to avoid new divergences in the subclass. It is the only part of the patterns which is not purely syntactic, but this is necessary since we want to take divergence into account.

The following main theorem now states the soundness of the patterns for the subtyping relations.

Theorem 1.

If C is a subclass of A obeying pattern P1 then $A \sqsubseteq_{wst} C$.

If C is a subclass of A obeying pattern P2 then $A \sqsubseteq_{ost} C$.

These patterns can be combined with approbriate[3] data-refinement rules: instead of requiring to leave old methods unchanged they may also be changed according to data refinement rules. This is still a correct change leading to a subtype. The proof of this is simple: changes according to data refinement rules induce failure-divergence refinement, and $A \sqsubseteq_{\mathcal{FD}} C1 \wedge C1 \sqsubseteq_{ost} C2 \Rightarrow A \sqsubseteq_{ost} C2$ (analogous for weak subtyping).

The rest of the paper is devoted to proving the above theorem. For the proof we show that, when constructed by a pattern, subclass and superclass are even more close to each other than defined by subtyping: the superclass *simulates* the subclass in a certain sense. Hence in the following we define two simulation relations inducing the two subtyping relations. Both relations are in spirit forward simulations. We do not consider backward simulations here since we are not interested in completeness. In fact, it is immediately clear that the patterns are not complete: there are as well valid subtypes which cannot be constructed according to the patterns.

For the simulation definitions we assume that the semantics of Definition 2 is used for classes, with the consequence that there are no τ-transitions except from $\perp$ to $\perp$ and the symbol ν does not occur as a label. We start with a simulation relation for weak subtyping.

Definition 7. *Class A* simulates *C* except for *N, $A \preceq_{\not N} C$, if there is a relation $\rho \subseteq Q_A \times Q_C$ such that*

1. $\forall\, q_C \in Q_C^{in}\ \exists\, q_A \in Q_A^{in} : (q_A, q_C) \in \rho$,
2. $\forall (q_A, q_C) \in \rho$ *the following holds:*
   - $stable(q_C) \Rightarrow stable(q_A)$,
   - $next(q_C) \setminus N = next(q_A)$,
   - $q_C\!\uparrow\ \Rightarrow q_A\!\uparrow$,
   - $\forall\, a \notin N$:
     $q_C \xrightarrow{a}_C q_C' \Rightarrow \exists\, q_A' : q_A \xrightarrow{a}_A q_A' \wedge (q_A', q_C') \in \rho$.

The idea of this simulation relation is simply to ignore all N-steps of the subclass. The next theorem shows the adequateness of using simulations except N for proving weak subtyping.

Theorem 2.
$A \preceq_{\not N} C \Rightarrow A \sqsubseteq_{wst}^{N} C$.

Proof: Assume $A \preceq_{\not N} C$ and let ρ be the simulation relation.

- Let $(\sigma, X) \in failures(C \setminus_r N)$, $\sigma = a_1 \ldots a_n, \sigma \in (\Sigma \setminus N)^*$. Then there is some $q_C^{in} \in Q_C^{in}, q_C \in Q_C, q_C$ stable and intermediate states $q_C^1 \ldots q_C^{n-1}$ such that $q_C^{in} \xrightarrow{a_1} q_C^1 \xrightarrow{a_2} \ldots \xrightarrow{a_{n-1}} q_C^{n-1} \xrightarrow{a_n} q_C$ and $X \subseteq refusals(q)$. From the simulation relation ρ we know the existence of states $q_A^{in}, q_A^1 \ldots q_A^{n-1}, q_A$ such that $q_A^{in} \xrightarrow{a_1} q_A^1 \xrightarrow{a_2} \ldots \xrightarrow{a_{n-1}} q_A^{n-1} \xrightarrow{a_n} q_A$ and $(q_A^{in}, q_C^{in}) \in \rho$, $\forall 1 \leq i < n$: $(q_A^i, q_C^i) \in \rho$ and $(q_A, q_C) \in \rho$.
  It follows that q_A is stable and since $C \setminus_r N$ refuses all N-actions we get $next(q_C) = next(q_C) \setminus N = next(q_A)$ and hence $(\sigma, X) \in failures(A)$.

[3] By approbriate we mean data refinement rules that induce failure-divergence refinement on the behavioural semantics. Such rules may vary with the formalisms used. In our case the data refinement rules of CSP-OZ are approbriate.

- Let $\sigma \in \mathit{divergences}(C \setminus_r N), \sigma \in (\Sigma \setminus N)^*$ be a minimal divergence of C. Analogous to the above case we can find q_C^{in}, q_C and q_A^{in}, q_A such that $q_C^{in} \overset{\sigma}{\Longrightarrow} q_C$, $q_C\!\uparrow$ and $(q_A^{in}, q_C^{in}) \in \rho$, $(q_A, q_C) \in \rho$. Hence $q_A\!\uparrow$ and thus $\sigma \in \mathit{divergences}(A)$.

$\square$

A similar formulation of simulation can be used for optimal subtyping. The N-steps of the subclass are now taken into account but we do not require a matching step of the superclass, instead states q' reached after executing new methods from some state q must be simulated by the matching partner of q.

Definition 8. *Class* A simulates C concealing N, $A \preceq_{(N)} C$, *if there is a relation* $\rho \subseteq Q_A \times Q_C$ *such that*

1. $\forall\, q_C \in Q_C^{in}\ \exists\, q_A \in Q_A^{in} : (q_A, q_C) \in \rho$,
2. $\forall (q_A, q_C) \in \rho$ *the following holds:*
   - $\mathit{stable}(q_C) \Rightarrow \mathit{stable}(q_A)$,
   - $\mathit{next}(q_C) \setminus N = \mathit{next}(q_A)$,
   - $q_C\!\uparrow \Rightarrow q_A\!\uparrow$,
   - $\forall\, a \notin N$:
     $q_C \xrightarrow{a}_C q'_C \Rightarrow \exists\, q'_A : q_A \xrightarrow{a}_A q'_A \wedge (q'_A, q'_C) \in \rho$,
     $\forall\, a \in N$:
     $q_C \xrightarrow{a}_C q'_C \Rightarrow (q_A, q'_C) \in \rho$.

Theorem 3.
$A \preceq_{(N)} C \Rightarrow A \sqsubseteq^N_{ost} C$.

Proof: Assume $A \preceq_{(N)} C$ and let ρ be the simulation relation.

- Let $(\sigma, X) \in \mathit{failures}(C \setminus_c N)$, i.e. there is some $(\sigma', X') \in \mathit{failures}(C), \sigma' = a_1 \dots a_n$, $\sigma' \in \Sigma^*$ and $\sigma = \sigma' \downarrow (\Sigma \setminus N)$, $X \subseteq X' \cup N$. Then there is some $q_C^{in} \in Q_C^{in}, q_C \in Q_C, q_C$ stable and intermediate states $q_C^1 \dots q_C^{n-1}$ such that $q_C^{in} \xrightarrow{a_1} q_C^1 \xrightarrow{a_2} \dots \xrightarrow{a_{n-1}} q_C^{n-1} \xrightarrow{a_n} q_C$ and $X' \subseteq \mathit{refusals}(q_C)$. From the simulation relation ρ we know the existence of states $q_A^{in}, q_A^1 \dots q_A^{n-1}, q_A$ such that $q_A^{in} \xrightarrow{a_1} q_A^1 \xrightarrow{a_2} \dots \xrightarrow{a_{n-1}} q_A^{n-1} \xrightarrow{a_n} q_A$ and $(q_A^{in}, q_C^{in}) \in \rho$, $\forall\, 1 \leq i < n$: $(q_A^i, q_C^i) \in \rho$ and $(q_A, q_C) \in \rho$. If $a_j \in N$ then $q_A^j = q_A^{j-1}$.
  It follows that q_A is stable and since $C \setminus_c N$ refuses all N-actions we get $\mathit{next}(q_C) = \mathit{next}(q_C) \setminus N = \mathit{next}(q_A)$ and hence $(\sigma, X) \in \mathit{failures}(A)$.
- Let $\sigma \in \mathit{divergences}(C \setminus_c N)$ be a minimal divergence, i.e. there is some $\sigma' \in \mathit{divergences}(C)$ such that $\sigma = \sigma' \downarrow (\Sigma \setminus N)$. Analogous to the above case we can find q_C^{in}, q_C and q_A^{in}, q_A such that $q_C^{in} \overset{\sigma}{\Longrightarrow} q_C$, $q_C\!\uparrow$ and $(q_A^{in}, q_C^{in}) \in \rho$, $(q_A, q_C) \in \rho$. Hence $q_A\!\uparrow$ and thus $\sigma \in \mathit{divergences}(A)$.

$\square$

These two simulation relations should be easily generalisable to classes with internal operations (besides the τ's between the divergent state $\perp$ that we have here) along the lines of [DBBS97].

The remaining part for proving Theorem 1 is to show that subclasses constructed by patterns can be simulated by the superclasses.

Theorem 4.
If C *is a subclass of* A *obeying pattern P1 and* $N = \{op_n \mid n \in I \setminus J\}$ *then* $A \preceq_{\not N} C$.

Proof: We construct a simulation relation except N.
Let $A = \big(\mathit{AState}, \mathit{AInit}, (\mathit{AOp}_j)_{j \in J}\big)$ and $C = \big(\mathit{CState}, \mathit{CInit}, (\mathit{COp}_i)_{i \in I}\big)$ be constructed from A by means of inheritance and obeying pattern P1. The simulation relation ρ is the following:

$(\perp, \perp) \in \rho$
$(\mathit{astate}, \mathit{astate} \oplus \mathit{extstate}) \in \rho$

where $astate \in AState$ is a state of the superclass and $extstate$ is an arbitrary state of the new attributes in C. The symbol $\oplus$ is used to denote disjoint union of the evaluation of old and new attributes. We show that it is indeed a simulation relation:

1. $\forall\, q_C \in Q_C^{in}$ there is some $q_A \in Q_A^{in}$ such that $(q_A, q_C) \in \rho$ holds since $CInit = AInit \wedge ExtInit$, and hence if $CInit$ holds for some state $state \oplus extstate$ then $AInit$ holds for $state$.
2. $\forall (q_A, q_C) \in \rho$ we show the following:
   - $stable(q_c) \Rightarrow stable(q_a)$ since the only instable state is $\perp$.
   - $next(q_C) \setminus N = next(q_A)$ since $COp_j = AOp_j$ and thus only depends on $state$.
   - $q_C\!\uparrow\, \Rightarrow q_A\!\uparrow$ since $\perp$ is the only divergent state.
   - if $q_C \xrightarrow{a}_C q'_C$, $a \notin N$ and $q_C = state \oplus extstate$, $q'_C = state' \oplus extstate'$, then a is also enabled in $state = q_A$ and $state \xrightarrow{a}_A state' = q'_A$ and $(q'_C, q'_A) \in \rho$.

□

A similar result holds for pattern P2.

Theorem 5.
If C is a subclass of A obeying pattern P2 and $N = \{op_n \mid n \in I \backslash J\}$ then $A \preceq_{(N)} C$.

Proof: Similar to the above proof. We use the same relation as simulation relation concealing N. One additional condition has to be checked.

- if $q_C \xrightarrow{a} q'_C$, $a \notin N$ and $q_C = state \oplus extstate$ then $q'_C = state \oplus extstate'$: $q'_C \neq \perp$ since no new method are enabled outside its precondition (P2.3) and $state$ is not changed by execution of a due to condition P2.2. By construction of ρ we have $(q_A, q'_C) \in \rho$.

□

Theorem 1 thus remains as a simple corollary. Summarising, these patterns now give as the possibility to obtain a valid weak or optimal subtype *by construction*.

5 Conclusion

In this paper we presented subtyping patterns for active classes with a behavioural semantics. Patterns are simple, mostly syntactic rules which, when applied during construction of subclasses via inheritance, guarantee correct subtypes. For proving soundness of the patterns we developed simulation relations similar to those used in data refinement theory. The patterns allow designers with no background in formal semantics to build classes which are subtypes of certain superclasses, thus inheriting their properties and being substitutable for them.

Related work. First of all there is a huge amount of work on behavioural subtyping relations for object-oriented formalisms. They differ in the semantic models used as a basis and in the defined relations. A rough classification could divide them into bisimulation and refinement-based approaches. Bisimulation based approaches easily achieve some form of substitutivity since they use an *equivalence* relation for subtyping. We consider this to be a too strong form since subtyping in its basic nature is not symmetric. The papers with refinement-based proposals on the other hand in most cases only discuss substitutability but do not give a formal proof. The relation most often used in this context is *extension*, developed for the process algebra LOTOS. Extension allows extension of functionality, but not only on new methods but also on old methods; it for instance allows new traces on old methods.

The discussion of related work concerning the specific topic of this paper can be split in two parts. The first one concerns the large amount of work on simulation relations for refinement, especially those showing the tight connections between data refinement in state-based and process refinement in behaviour-based systems [He89,Jos88,WM90,DBBS97,SD97]. The ideas of these papers are clearly the basis for our simulation relations which adapt this principle concept to the case of behavioural subtyping relations. Here we had to cope with the existence of additional operations in the subclass and, depending on the way they are treated in the subtyping relations, different kinds of simulation relations had to be developed. A further adaption of ideas from [DBBS97] concerning weak refinement could generalise our results to classes with internal operations.

The second area are patterns for subtyping relations. The work of van der Alst and Basten [vdAB97] presents transformation rules on Petri nets which construct subtypes, where the subtyping relations are based on branching bisimilarity. Saake et al. [SHJ$^+$94] give graph manipulations on transition systems describing life-cycles of objects that guarantee certain so called "inheritance conditions" to hold. Similar kinds of conditions for a simple form of statecharts are developed in [EE94].

Patterns which are similar to ours are also employed in UNITY's superposition operator [CM88]. The setting there is not object-orientation, nevertheless the goal is similar: extend a program in a conservative way, not disturbing any of the properties holding beforehand. In fact, the idea of conservative extension appears in many theories and maybe subtyping can just be seen as a form of conservative extension in object-orientation.

References

[Ame91] P. America. Designing an object-oriented programming language with behavioural subtyping. In J.W. de Bakker, W.P. de Roever, and G. Rozenberg, editors, *REX Workshop: Foundations of Object-Oriented Languages*, number 489 in LNCS. Springer, 1991.

[BBSDS97] H. Bowman, C. Briscoe-Smith, J. Derrick, and B. Strulo. On behavioural subtyping in LOTOS. In H. Bowman and J. Derrick, editors, *Formal methods for open object-based distributed systems*, pages 335 – 351. Chapman & Hall, 1997.

[BCP99] C. Balzarotti, F. De Cindio, and L. Pomello. Observation equivalences for the semantics of inheritance. In P. Ciancarini and R. Gorrieri, editors, *FMOODS '99, Formal methods for open object-based distributed systems*, 1999.

[CM88] K.M. Chandy and J. Misra. *Parallel program design: a foundation*. Addison Wesley, 1988.

[Cus91] E. Cusack. Refinement, conformance and inheritance. *Formal Aspects of Computing*, 3:129 – 141, 1991.

[DBBS97] J. Derrick, E. Boiten, H. Bowman, and M. Steen. Weak refinement in Z. In J. Bowen and M. Hinchey, editors, *ZUM'97: The Z Formal Specification Notation*, volume 1212 of *LNCS*, pages 369–388, 1997.

[DL96] K. K. Dhara and G. T. Leavens. Forcing behavioral subtyping through specification inheritance. In *Proceedings of the 18th International Conference on Software Engineering, Berlin, Germany*, pages 258–267. IEEE Computer Society Press, March 1996.

[EE94] J. Ebert and G. Engels. Observable or invocable behaviour - you have to choose! Technical Report tr94-38, University of Leiden, 1994.

[Fis97] C. Fischer. CSP-OZ: A combination of Object-Z and CSP. In H. Bowman and J. Derrick, editors, *Formal Methods for Open Object-Based Distributed Systems (FMOODS '97)*, volume 2, pages 423–438. Chapman & Hall, 1997.

[Fis00] C. Fischer. *Combination and Implementation of Processes and Data: From CSP-OZ to Java*. PhD thesis, University of Oldenburg, 2000.

[FW00] C. Fischer and H. Wehrheim. Behavioural subtyping relations for object-oriented formalisms. In T. Rus, editor, *AMAST 2000: International Conference on Algebraic Methodology And Software Technology*, number 1816 in Lecture Notes in Computer Science. Springer, 2000.

[GHJV85] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of reusable object-oriented software.* Addison Wesley, 1985.

[He89] J. He. Process simulation and refinement. *Formal Aspects of Computing*, 1(3):229–241, 1989.

[HK99] D. Harel and O. Kupferman. On the inheritance of state-based object behaviour. Technical Report MCS99-12, The Weizmann Institute of Science, Israel, 1999.

[Hoa85] C. A. R. Hoare. *Communicating Sequential Processes.* Prentice-Hall, 1985.

[Jos88] M.B. Josephs. A state-based approach to communicating processes. *Distributed Computing*, 3:9–18, 1988.

[LW94] B. Liskov and J. Wing. A behavioural notion of subtyping. *ACM Transactions on Programming Languages and Systems*, 16(6):1811 – 1841, 1994.

[Nie95] O. Nierstrasz. Regular types for active objects. In O. Nierstrasz and D. Tsichritzis, editors, *Object-oriented software composition*, pages 99 – 121. Prentice Hall, 1995.

[Ros97] A. W. Roscoe. *The Theory and Practice of Concurrency.* Prentice-Hall, 1997.

[SD97] G. Smith and J. Derrick. Refinement and verification of concurrent systems specified in Object-Z and CSP. In M. Hinchey and Shaoying Liu, editors, *Int. Conf. of Formal Engineering Methods (ICFEM)*, pages 293–302. IEEE, 1997.

[SHJ+94] G. Saake, P. Hartel, R. Jungclaus, R. Wieringa, and R. Feenstra. Inheritance conditions for object life cycle diagrams. In U. Lippeck and G. Vossen, editors, *Workshop Formale Grundlagen für den Entwurf von Informationssystemen*, number 03/94. Uni Hannover, 1994.

[Smi00] G. Smith. *The Object-Z Specification Language.* Kluwer Academic Publisher, 2000.

[vdAB97] W.M.P. van der Aalst and T. Basten. Life-cycle inheritance - a Petri-net-based approach. In P. Azema and G. Balbo, editors, *Application and Theory of Petri Nets*, number 1248 in LNCS, pages 62–81. Springer, 1997.

[Wal99] J. Waldo. The Jini architecture for network-centric computing. *Communications of the ACM*, 42(7):76–82, 1999.

[Weh00] H. Wehrheim. Behavioural subtyping and property preservation. In *FMOODS'00: Formal Methods for Open Object-Based Distributed Systems*, 2000. to appear.

[WM90] J. C. P. Woodcock and C. C. Morgan. Refinement of state-based concurrent systems. In *Proceedings of VDM Symposium 1990*, volume 428 of *LNCS*, pages 340–351. Springer-Verlag, 1990.

[WZ88] P. Wegner and B. Zdonik. Inheritance as an incremental modification mechanism or What like is and isn't like. In *ECOOP'88: European Conference on Object-Oriented Programming*, volume 322 of *Lecture Notes in Computer Science*. Springer, 1988.