BEHAVIOURAL SUBTYPING AND PROPERTY PRESERVATION

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

Abstract Inheritance is one of the key features in object-oriented design and analysis. It especially supports an *incremental* development by allowing to stepwise add new functionality to an existing system design.
When using a subclass which is a specialisation of a certain superclass, the question arises which of the superclass' properties still hold for the subclass. We investigate this topic for three *behavioural subtyping* relations, which formalise the subtype - supertype relationship among classes on the basis of process algebra correctness relations. According to the degree of change allowed by the subtyping relations, safety and liveness properties of the superclass are preserved up to different extents.

Keywords: Behavioural subtyping, process algebra, refinement, inheritance.

1. INTRODUCTION

Inheritance is the one of the key issues for the success of object-oriented analysis and design methods as well as programming languages. The main purpose of inheritance is to support the structuring of specifications and code, and to allow re-use of already written parts. Inheritance can furthermore be applied in the *incremental development* of large systems, starting with a small basic system description on which additional functionality is stepwise added until the complete system has been designed. Such a development process would start with a superclass, capturing the basic requirements on the system, and derive new subclasses by inheritance which stepwise add new features to the existing functionality. However, this approach is in general not error-free: the addition of new features may interfere with the old ones in an undesired way. In the telecommunications area this phenomenon is known as *feature interaction*. Inheritance in general does not provide any help in avoiding feature interactions; in fact, in many applications of inheritance, as for instance simple

code re-use, it is not desired to restrict the changes allowed in the subclass in any way (see [13] for a classification of different forms of inheritance).

A relationship among subclass and superclass which requires the subclass to reflect the behaviour of the superclass to a certain degree is *subtyping*. Subtyping in object-oriented formalisms lifts the usual notion of subtype to the level of objects. Subtypes should always fulfil the principle of *type substitutability* [19]: any changes in the subtype should be transparent to users of the supertype. This is also the requirement that we have on an incremental design: any addition of a new feature to a class should keep the old functionality unchanged. Simple code-reuse often does not fulfil the principle of substitutability, and in fact this is most often not even intended. The importance of subtyping concepts for object-oriented formalisms is witnessed by an increasing concern in this topic: as an example, the new UML version 1.3 [15] discusses a special subtyping category for statechart refinement; and Sun's new JINI architecture [18] 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.

Subtyping definitions are usually signature-based, i.e. compare operations of sub- and supertype according to their signatures. However, signatures alone cannot guarantee substitutability since the *semantics* of methods may change completely while retaining the signature. *Behavioural subtyping* as introduced by for instance [12, 7, 1] overcomes this problem by comparing methods of sub- and supertype according to their pre- and postconditions and their preservation of global constraints on the type. A second, different, approach, which we follow in this paper (also with the name behavioural subtyping), is taken by directly comparing the *dynamic behaviour* (ordering of method execution) of classes, not their methods in isolation. This approach takes a view on classes as being *active* entities. In concurrent object-oriented languages active objects have their own thread of control and often have to obey particular protocols in order to behave properly. A number of authors have proposed such behavioural subtyping relations [6, 14, 4, 3, 17, 2, 9, 8] for object-oriented formalisms, most often based on some process algebra correctness relation. These relations sometimes come with an alternative *testing characterisation*, which formalises the degree of substitutability obtained by subtypes. The testing characterisations can be used to figure out the appropriate application domain of the particular subtyping relation: is it only sufficient for objects with a single client at a moment or also correct for shared mutable objects.

Valid subtyping is so far defined as "obtaining a sufficient degree of substitutability". A different view on correctness of subtypes is taken when instead the *preservation of properties* of supertypes in their subtypes is considered. In this paper we will be concerned with this view on behavioural subtyping: we will investigate up to which extent *properties* of a superclass are preserved un-

der behavioural subtyping. Properties may for instance be safety requirements on the methods of the superclass, or liveness requirements, guaranteeing the availability of certain services to clients of the class. Ideally, such properties should still hold for subclasses, extending the superclass with additional features. This would allow us to avoid the re-verification of properties on a subtype which have already been proven to hold on the supertype. We will study the preservation of safety and liveness properties under three behavioural subtyping relations, two already appearing in [8], the third a new one.

As it turns out, the class of preserved properties of a particular subtype can be seen as a *characteristic* of the subtyping relation. This, together with the testing characterisations of [8], gives a second view on subtyping relations, and thus sheds some more light on the question of which subtyping relation is the most appropriate one for a given application domain.

We envisage the following use of subtyping relations in the development process: for every class, on which we intend to further add some functionality by means of inheritance, we figure out its area of application: class with single access or with shared access, and depending on this, the appropriate subtyping relation guaranteeing substitutability for the application area can be chosen. Afterwards all subclasses derived from this class have to be checked whether they are correct with respect to the chosen behavioural subtype. The contribution of this paper is to show which properties of the supertype now also hold for the subtype. This avoids re-verification of properties for every new subtype which is created and thus supports *modular reasoning.*

The paper is structured as follows: The next section defines the technical basis (labelled transition systems and refinement relations), Section 3 introduces behavioural subtyping and gives some examples. The following section then starts the discussion on property preservation and Section 5 finally gives the results.

2. DEFINITIONS

Most of the behavioural subtyping relations focusing on the dynamic behaviour of classes are based on some process algebra correctness relation, like failures refinement or bisimulation. The application area for process algebras is the description of distributed communicating processes, thus they are a reasonable choice as a basis for behavioural subtyping for active objects. The three relations we consider in this paper are all based on CSP theory [10, 16].

We start with the definition of the relevant concepts underlying our subtyping relations and correctness criteria. Labelled transition systems (short LTS) are used for describing the behaviour of an object (or more precise: of its class), the methods that are called on *and* by the object and their order of execution. We view an LTS as describing the *behavioural type* of an active

object. The semantical basis for the behavioural subtyping relations and the correctness checks is the *failures model* of the process algebra CSP. For this the communication events of CSP are identified with method invocations.

We assume Σ to be a set of methods, in the following also referred to as *actions*. We furthermore have two special *invisible actions*: $\tau, \nu \notin \Sigma$ and let $\Sigma_{\tau\nu} = \Sigma \cup \{\tau, \nu\}$. Both actions stand for unobservable entities: the symbol τ plays the usual role of an *internal* action of an object, whose execution is under the control of the object alone; the symbol ν stands for an action which is *invisible* to a particular client of an object, but is not under control of the object alone. It may have to be executed together with another client of the object.

Definition 2.1 *A* labelled transition system *(LTS) is a tuple* $T = (Q, \rightarrow, q_0)$ *such that*

- Q *is a set of states,*
- $\rightarrow \subseteq Q \times \Sigma_{\tau\nu} \times Q$ *is a transition relation and*
- $q_0 \in Q$ *is the initial state.*

A labelled transition system of an object describes the possible states of an object (identified by particular values of its attributes), the transitions between states (which methods are enabled in a state and how the state changes with an execution of a method) and its initial state.

We write $q \xrightarrow{a} q'$ if $(q, a, q') \in \rightarrow$. Let $\sigma \in \Sigma_{\tau\nu}^*$ be a sequence of actions and $A \subseteq \Sigma$ a set of actions. The *projection of* σ *on* A, $\sigma \downarrow A$, is the trace where all occurrences from events not in A are removed. The *alphabet* of an LTS T, $\alpha(T)$, is the set of actions occurring as labels in the transition relation.

An LTS describes the behaviour of an object. When we compare two objects with respect to their behaviour, we look at the *traces* they may execute (sequences of methods) and their *failures* (what methods are blocked after a particular trace). Traces and failures are derived from transition systems.

Definition 2.2 *Let* $T = (Q, \rightarrow, q_0)$ *be a labelled transition system,* $q, q' \in Q, a_i \in \Sigma_{\tau\nu}$ *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* $q = q_0$, $q_i \xrightarrow{a_{i+1}} q_{i+1}$ *and* $q_n = q'$.
- $q \stackrel{\sigma}{\Longrightarrow} q'$ *iff there is a trace* $t \in \Sigma_{\tau\nu}^*$ *such that* $q \xrightarrow{t} q'$ *and* $\sigma = t \downarrow \Sigma$.
- *The set of* traces *of* T *is*

$$traces(T) := \{\sigma \in \Sigma^* \mid \exists q \in Q : q_0 \stackrel{\sigma}{\Longrightarrow} q\} .$$

- *A state is* stable *if no* τ *transitions are possible:* q *stable iff* $q \xrightarrow{\tau}\!\!\!\!\!\!/\;\;$.

The set of enabled actions *of a state* $q \in Q$ *is*

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

its maximal refusals *are* $refusals(q) := \Sigma \setminus next(q)$

- *The set of* failures *of* T *is*

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

Note, that failures always have subset-closed refusal sets. Failure sets only record the refusals at *stable* states since τ actions (whose execution is under control of the object alone) might lead to states which refuse something different. Refinement relations of CSP relate transition systems with respect to their traces and failures.

Definition 2.3 *A labelled transition system* I *is a* trace-refinement *of an LTS* S *(denoted* $S \sqsubseteq_{\mathcal{T}} I$*) iff* $traces(I) \subseteq traces(S)$*, it is a* failures-refinement *(*$S \sqsubseteq_{\mathcal{F}} I$*) iff* $failures(I) \subseteq failures(S)$.

Some remark concerning the notation: in CSP, the symbol $\sqsubseteq$ is used in a somewhat unusual direction, since on the left hand side we find the specification, i.e. the more nondeterministic system, whereas on the right hand side the more refined implementation is found. The subtyping relations defined later in the paper are all based on refinement, therefore we also use this direction there: the subtype will always stand on the right hand side.

These two refinement relations are the basis for the subtyping relations introduced in the next section and the correctness criteria defined in Section 4. We will not treat divergence (livelocks) in this paper, but it can be easily incorporated into the subtyping relations presented here.

Besides the two refinement concepts, we will furthermore need two operators on labelled transition systems: restriction and concealment. Restriction is a standard process algebra operator (from CCS), concealment is a form of hiding, which however just makes some set of actions invisible, but not internal.

Definition 2.4 *Let* $T = (Q, \rightarrow, q_0)$ *be a labelled transition system and* $A \subseteq \Sigma$.

- *The* restriction of A in T, $T \setminus_r A$, *is defined as* $(Q, \rightarrow', q_0)$ *with*

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

- *The* concealment of A in T, $T \setminus_c A$, *is defined as* $(Q, \rightarrow', q_0)$ *with*

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

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 afterwards under control of the object alone. Hiding may usually introduce new non-stable states, whereas concealment does not.

3. BEHAVIOURAL SUBTYPING RELATIONS

We start the introduction into subtyping relations with a discussion of three examples. For all three examples we give a labelled transition system of a superclass describing the basic behaviour of the system (in solid lines) and afterwards look at some possible extension made in a subclass (in dashed lines). The initial state of the LTS is marked by a circle. Due to the augmentation of the alphabet of the LTS during extension (new methods), traces or failures refinement cannot be directly used as a subtyping relation; instead we have to find a reasonable way of hiding the new methods during the comparison.

The first example is from the telecommunication area: the LTS of the superclass describes the basic behaviour of a (simplified) telephone interface to a *single* user, starting with *off_hook* followed by a *dial_tone* and the *dial*ing of a number. Afterwards a *response* or *no_response* from the called side follows, in case of a response the user of the telephone will *talk* to someone (and the telephone has to transmit this). At all times, the user is free to hang up the telephone (*on_hook*).

The solid states and arrows in Figure 1 show the LTS of the class for a basic telephone interface. This class can be the basis for further extensions with different features, as for instance call forwarding, call screening or voicemail. The extension we look at here is concerned with call screening, namely the possibility of entering telephone numbers of people from which the user does not want to accept calls: the new feature has to be chosen after *off_hook* (*screen_call*), the number has to be entered (*type_number*), is again *display*ed and has to be *acknowledge*d in order to be inserted in the list of screened numbers. The new feature is shown in Figure 1 in dashed lines.

No feature interactions occur in this extension: the telephone company can be sure that all users are able to use the old service as before, without noticing any difference. The basic requirement for this extension "no new behaviour on the old service" is fulfilled[1]. This is the fundamental requirement for all subtyping relations, and can be formalised with the help of the restriction operator:

[1]Not all kind of features in the telecommunication area have this correctness requirement; a lot of features intentionally want to change the old behaviour (e.g. teen-line) and then a feature interaction occurs when the old service remains available.

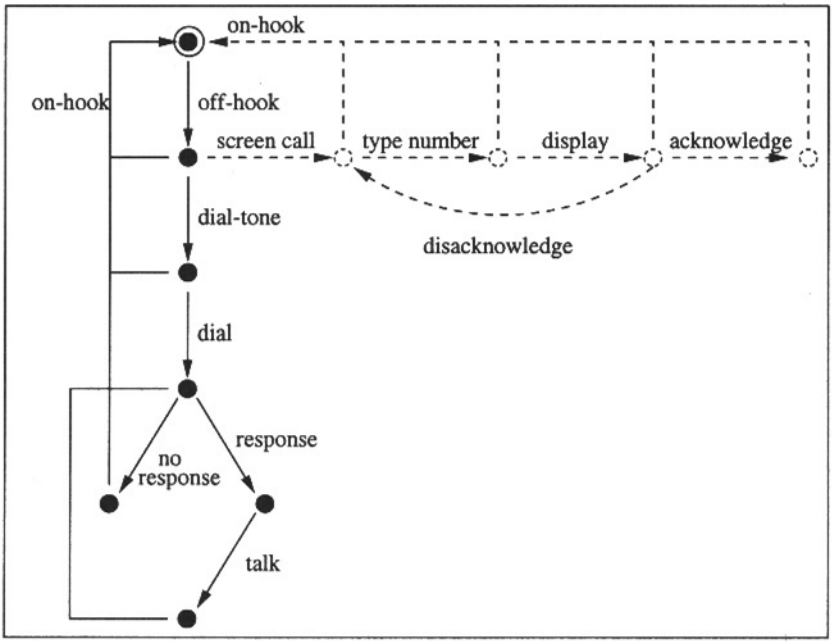

Figure 1 Basic telephone service (+ call screening input)

Definition 3.1 (Weak subtyping) *Let U, O be LTS' and $N = \alpha(U) \setminus \alpha(O)$. U is a* weak subtype *of O (denoted $O \sqsubseteq^{N}_{wst} U$) iff*

$$O \sqsubseteq_{\mathcal{T}} U \setminus_r N \text{ and}$$
$$O \sqsubseteq_{\mathcal{F}} U \setminus_r N .$$

Weak subtyping compares sub- and superclass by completely neglecting to look at the new service. The extension in the above example is a weak subtype of the superclass.

The next example and its extension seems to be of a similar kind but reveals the need for a different subtyping relation. Figure 2 shows the transition system of a simple till: the customer inserts a card into the till and may then choose to stop the interaction or make some withdrawal of money. Then (s)he has to type her/his pin (personal identification number); when it is correct, the amount of money can be chosen and is delivered, when incorrect, another try can be made or the transaction can be stopped. The new feature which is added to the basic till is the printing of the balance. Once the card has been inserted, the user may choose to print the balance and in this case no pin has to be given (the data on the card is sufficient for this). After printing the user may stop the transaction or make some withdrawal. The (dashed-line) extension in Figure 2

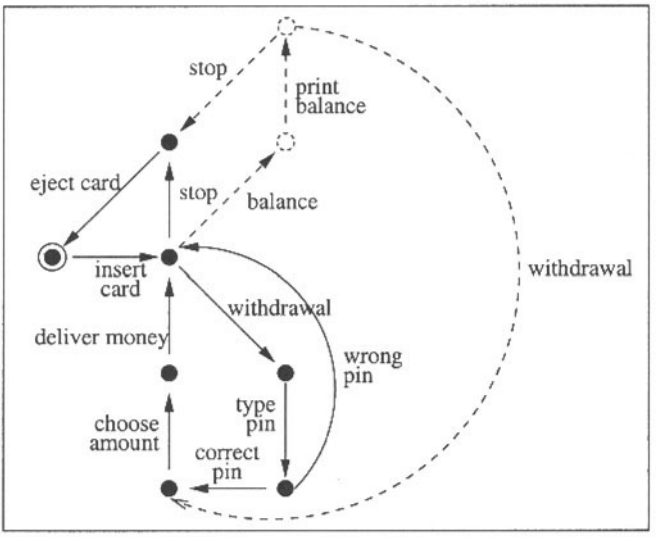

Figure 2 Till (+ balance printing)

for this additional feature is obviously wrong: the typing of the pin can now be circumvented when the printing is chosen first.

With regard to the above defined subtyping relation, the extended till is a valid weak subtype, although the new feature interacts with the old service in an undesired way. This leads us to the definition of a second subtyping relation, which rules out such undesired extensions:

Definition 3.2 (Safe subtyping) *Let U, O be LTS' and $N = \alpha(U) \setminus \alpha(O)$. U is a* safe subtype *of O (denoted $O \sqsubseteq^N_{sst} U$) iff*

$$O \sqsubseteq_{\mathcal{T}} U \setminus_c N \text{ and}$$
$$O \sqsubseteq_{\mathcal{F}} U \setminus_r N .$$

This relation requires that even if a client uses a new service, no new possibilities of using the old service arise (condition on traces). Furthermore, like before, we require that the old service still works correct. The above extension of the simple till is not a safe, only a weak subtype. In this safety critical application, especially in the case, where the interface explicitly informs the user about the new service (this is what tills do), weak subtyping is not sufficient.

A last example shows why we are still not at the end of defining relations. The following example is different from the previous ones in that it allows more than one client to access the services of the class at one moment. The example, given in Figure 3, is the interface to a simple printer, which accepts requests

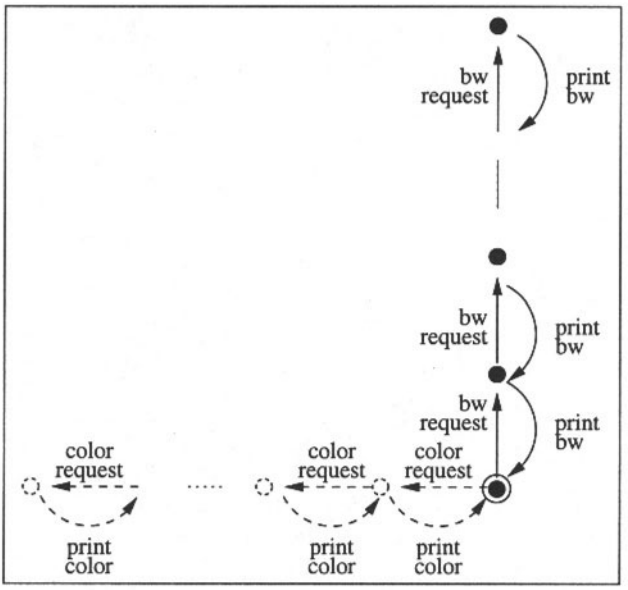

Figure 3 Black-and-white (+ colour) printer

for printing black-and-white pictures, *bw_request* (up to a certain amount depending on its internal store), and successively prints them (*print_bw*).

The new feature to be added is the printing of colour pictures. The (dashed line) extension is incorrect in that it might at some times block requests for black-and-white prints (after some colour prints have been requested but not yet printed). Thus, as soon as another client is additionally using the new feature of the interface, users of the old service might indeed notice a difference. The extension is however both a weak and a safe subtype. Instead, for *shared objects* the following subtyping relations is necessary to capture the desired substitutability:

Definition 3.3 (Optimal subtyping) *Let U, O be LTS' and $N = \alpha(U) \setminus \alpha(O)$. U is an* optimal subtype *of O (denoted $O \sqsubseteq_{ost}^{N} U$) iff*

$$O \sqsubseteq_{\mathcal{T}} U \setminus_c N \text{ and}$$
$$O \sqsubseteq_{\mathcal{F}} U \setminus_c N .$$

This definition defines a comparison which additionally involves an inspection of the refusals in the *new* part of the LTS. For the users of the old service, no new refusals (corresponding to a blocking of requests) are allowed. A correct extension of the printer interface would have to serve the requests for black-and-white and colour prints concurrently.

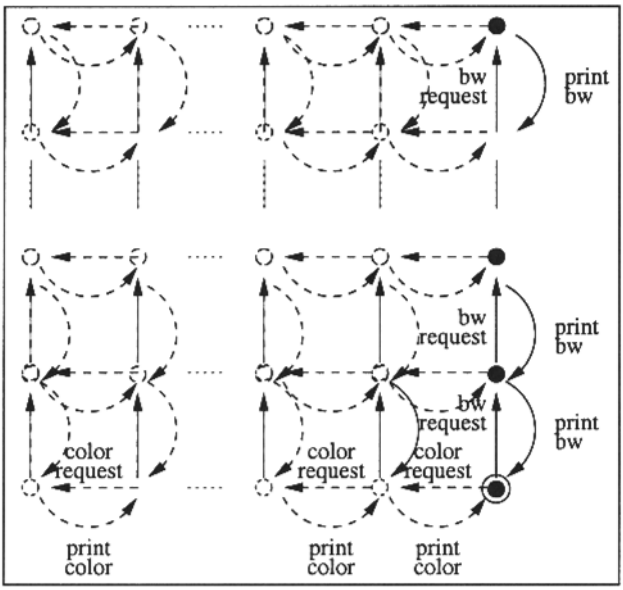

Figure 4 Correct extension of printer

Optimal subtyping is the strongest of the three behavioural subtyping relations.

Proposition 3.1 *Let U, O be LTS', $N = \alpha(U) \setminus \alpha(O)$. Then*

$$O \sqsubseteq^N_{ost} U \Rightarrow O \sqsubseteq^N_{sst} U \text{ and}$$
$$O \sqsubseteq^N_{sst} U \Rightarrow O \sqsubseteq^N_{wst} U .$$

A correct extension of the printer is shown in Figure 4: the new feature can be concurrently executed with the old one. For a client of the black-and-white printing feature the same service as before is possible.

Summarising, we now have three subtyping relations for extension of functionality: two for classes with single access, the first one guarantees a minimal correctness on the old service, the second one also guarantees that the new service cannot be used to achieve new effects on the old features, and the third subtyping relation guarantees this also for shared mutable objects.

For two of the subtyping relations (weak and optimal), corresponding testing characterisations can be found in [8]. The definition of safe subtyping is new[2].

[2]A subtyping relation with the same name appears in [8], but we prefer to call the relation defined here "safe", since it is safe in the sense of preserving all safety properties.

4. PROPERTIES AND INHERITANCE

The main issue of this paper is the relationship between properties which hold for a superclass and for its subtypes. Ideally, subtypes should preserve all properties of the superclass.

In the sequel, we take a CSP-like view on property specification: in CSP theory properties are expressed as CSP processes (or directly LTS') and are checked on some specification by a comparison of property and specification with respect to their traces (then we speak of a *safety* property) or their failures (a *liveness* property). A property is therefore a process or an LTS describing every allowed behaviour and it is checked whether the behaviour of the implementation is a subset of this allowed behaviour: We check whether

$$P \sqsubseteq_{\mathcal{T}} S \quad \text{or}$$
$$P \sqsubseteq_{\mathcal{F}} S$$

holds for safety or liveness properties P, respectively, and objects S. Since all refinement relations are transitive, we can always make the following deduction:

$$P \sqsubseteq_{\mathcal{T}} S \sqsubseteq_{\mathcal{T}} I \Rightarrow P \sqsubseteq_{\mathcal{T}} I \quad \text{or} \quad P \sqsubseteq_{\mathcal{F}} S \sqsubseteq_{\mathcal{F}} I \Rightarrow P \sqsubseteq_{\mathcal{F}} I\,,$$

that is, all properties proven for S also hold for some refinement I.

It would be highly desirable to have a similar reasoning for superclasses and their subtypes: prove a property for a superclass and know that it also holds for all of its subclasses, which are behavioural subtypes. This is especially important for an *incremental design*, which uses inheritance to subsequently add new features to already existing and provably correct superclasses. However, before we can find out, which kind of properties are preserved for which subtype, we have to make clear what we mean by "a subclass has the same properties". Again, the usual refinement relations cannot be plainly used here since inheritance usually introduces *new* methods in the subclass (extension of functionality). A subclass may have traces over a larger alphabet than the superclass and may therefore fail to be a trace or failures refinement of some property process of which the superclass was a correct refinement. Nevertheless, the subclass may in a broader sense have the same property.

We explain this idea on the till example of the last section. The basic till is correct with respect to the safety property "money is only delivered when the correct pin has been typed in". Formalised in CSP, the property on the simple till can be formulated by a process, whose traces involve all possible orderings of actions from Σ in which every method *deliver_money* is preceded by an action *correct_pin* (interleaving, `|||`, of the correct order with all possible orderings of the other actions, `CHAOS`):

```
Prop = correct_pin -> deliver_money -> Prop
```

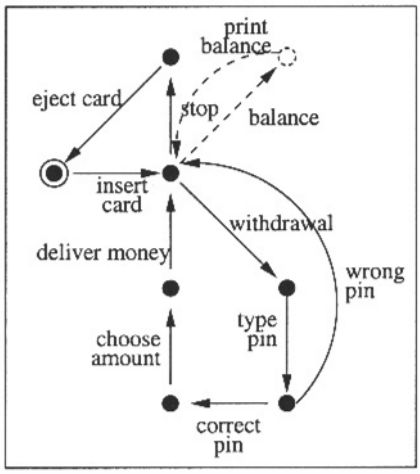

Figure 5 Correct extension of till

```
P = (CHAOS(Σ\{correct_pin,deliver_money}) ||| Prop)
```

The simple till is a trace refinement of such a safety tester process, the *wrong* extension is not. This so far is a first observation on property preservation in weak subtypes. Now consider a *correct* extension (as shown in Figure 5). This correct extension is a safe subtype of the basic till and also a trace refinement of P. Is it now a trace refinement of all processes *P*, of which the basic till is? The answer is no, since we may also use tester processes, which already fix orderings between *new* methods of the subclass, for instance: *print_balance* always before *balance*. The basic system trivially fulfils this requirement since *none* of these actions occur. Nevertheless, the property does not hold for the extended till, thus we have no property preservation here.

Hence we have to be more precise about *what* the subclass is expected to preserve, and what it means for a subclass to have the same properties as the superclass. What it should preserve are all restrictions formulated on "old" methods, i.e. those of the superclass. Any property talking about methods not present in the superclass clearly cannot be inherited, although the superclass may trivially fulfil it. There are two possibilities of formulating the instead needed "preservation on old methods": we only check the property on traces over the old methods or we check it on all traces, but by only looking at old methods. We call the former type of preservation "weak" and the more general form "strong fulfilment". The first extension of the till only weakly preserves the safety property (as long as the old service is used, usage is safe), the second extension also strongly fulfils the safety requirement.

Definition 4.1 (Safety) *Let P, S be LTS', P describing some safety property, S an arbitrary object, and let $A \subseteq \Sigma$ be a set of actions (the "old" methods).*
S weakly fulfils the safety properties of *P* on *A* ($P \sqsubseteq^A_{w\mathcal{T}} S$) *iff*

$$traces(S) \cap A^* \subseteq traces(P) \cap A^* .$$

S strongly fulfils the safety property of *P* on *A* ($P \sqsubseteq^A_{s\mathcal{T}} S$) *iff*

$$traces(S) \downarrow A \subseteq traces(P) \downarrow A .$$

Strong fulfilment guarantees that the restrictions on orderings of actions as specified by the safety property *P* are met in an object *S*, which possibly has more methods than those of *A*. Similarly fulfilment of liveness properties can be defined.

Definition 4.2 (Liveness) *Let P, S be LTS', P describing some liveness property, S an arbitrary object, and let $A \subseteq \Sigma$ be a set of actions (the "old" methods).*
S weakly fulfils the liveness properties of *P* on *A* ($P \sqsubseteq^A_{w\mathcal{F}} S$) *iff*

$$(\sigma, X) \in failures(S), \sigma \in A^* \\ \Rightarrow \exists (\sigma, Y) \in failures(P) \text{ such that } X \cap A = Y \cap A .$$

S strongly fulfils the liveness properties of *P* on *A* ($P \sqsubseteq^A_{s\mathcal{F}} S$) *iff*

$$(\sigma, X) \in failures(S) \\ \Rightarrow \exists (\sigma', Y) \in failures(P) \text{ such that } \sigma \downarrow A = \sigma' \downarrow A \\ \text{and } X \cap A = Y \cap A .$$

The condition on refusal sets guarantees that, with respect to *A*, the same set of actions are blocked. Strong preservation always implies weak preservation.

Proposition 4.1 *Let P, O be LTS', $A \subseteq \Sigma$.*

$$P \sqsubseteq^A_{s\mathcal{T}} O \Rightarrow P \sqsubseteq^A_{w\mathcal{T}} O \quad \text{and} \\ P \sqsubseteq^A_{s\mathcal{F}} O \Rightarrow P \sqsubseteq^A_{w\mathcal{F}} O$$

The above question "does subtyping preserve properties" can now be formulated as: if $P \sqsubseteq_{\mathcal{T}/\mathcal{F}} O_1$ and O_2 is a subtype of O_1, does O_2 weakly or strongly fulfil property *P* on $\alpha(O_1)$?

5. RESULTS

Next we investigate whether our three subtyping relations weakly or strongly preserve safety and liveness properties. Afterwards we discuss property preservation as another characterisation of subtyping relations.

5.1. PROPERTY PRESERVATION

In the following we always assume $N = \alpha(U) \setminus \alpha(O)$ to be the set of new methods, and often omit N as an index to the subtyping relations. We start with the treatment of safety properties. Weak subtypes weakly preserve safety properties.

Theorem 1 *Let O, U be LTS' such that $O \sqsubseteq_{wst} U$ and let P be an LTS describing a safety property. Then*

$$P \sqsubseteq_{\mathcal{T}} O \Rightarrow P \sqsubseteq_{w\mathcal{T}}^{\alpha(O)} U .$$

Proof: Assume (i) $P \sqsubseteq_{\mathcal{T}} O$, i.e. $traces(O) \subseteq traces(P)$ and (ii) $O \sqsubseteq_{\mathcal{F}} U \setminus_r N$, i.e. in particular $traces(U \setminus_r N) \subseteq traces(O)$. By definition of $\setminus_r$, we get the following: $traces(U \setminus_r N) = traces(U) \cap \alpha(O)^*$. Hence $traces(U) \cap \alpha(O)^* \subseteq traces(P)$, and thus also $traces(U) \cap \alpha(O)^* \subseteq traces(P) \cap \alpha(O)^*$. □

For the simple till and its first extension (a weak subtype), we therefore obtain the result that the subclass weakly preserves all safety properties of the simple till. As the example has also shown, weak subtyping does not guarantee strong preservation of safety properties. But safe and optimal subtyping do:

Theorem 2 *Let O, U be LTS' such that $O \sqsubseteq_{sst} U$ and let P be an LTS describing a safety property. Then*

$$P \sqsubseteq_{\mathcal{T}} O \Rightarrow P \sqsubseteq_{s\mathcal{T}}^{\alpha(O)} U .$$

Proof: Assume (i) $P \sqsubseteq_{\mathcal{T}} O$, i.e. $traces(O) \subseteq traces(P)$ and (ii) $O \sqsubseteq_{sst} U$, i.e. $traces(U \setminus_c N) \subseteq traces(O)$. By definition of concealment we get $traces(U \setminus_c N) = traces(U) \downarrow \alpha(O)$ and hence $traces(U) \downarrow \alpha(O) \subseteq traces(P)$. Since $traces(O) = traces(O) \downarrow \alpha(O)$, we get $traces(O) \downarrow \alpha(O) \subseteq traces(P) \downarrow \alpha(O)$, which all in one gives the desired result. □

The analogous result for optimal subtypes is a corollary of the last theorem and Proposition 3.1. With respect to safety properties, safe and optimal subtyping are thus equally suitable. The second (correct) extension of the simple till is a safe subtype and therefore strongly preserves all safety properties.

Considering liveness properties, we get the following two results. Weak subtyping weakly preserves liveness properties:

Theorem 3 *Let O, U be LTS' such that $O \sqsubseteq_{wst} U$ and let P be an LTS describing a liveness property. Then*

$$P \sqsubseteq_{\mathcal{F}} O \Rightarrow P \sqsubseteq_{w\mathcal{F}}^{\alpha(O)} U .$$

Proof: Assume (i) $failures(O) \subseteq failures(P)$ and (ii) $O \sqsubseteq_{wst} U$, i.e. $O \sqsubseteq_{\mathcal{F}} U \setminus_r N$. By definition of $\setminus_r$ we have

$failures(U \setminus_r N) = \{(\sigma, X) \mid \exists Y : (\sigma, Y) \in failures(U), \sigma \in \alpha(O)^*, X \subseteq Y \cup N\}$.
The rest follows by a simple application of definitions:

$$\begin{array}{rl} (\sigma, X) \in failures(U), \sigma \in \alpha(O)^* & \\ \Downarrow & \text{Definition of } \setminus_r \\ (\sigma, X \cup N) \in failures(U \setminus_r N) & \\ \Downarrow & O \sqsubseteq_{wst} U \\ (\sigma, X \cup N) \in failures(O) & \\ \Downarrow & P \sqsubseteq_{\mathcal{F}} O \\ (\sigma, X \cup N) \in failures(P) & \end{array}$$

and furthermore $X \cap \alpha(O) = (X \cup N) \cap \alpha(O)$ holds. □
With a similar kind of reasoning we can prove weak preservation of liveness properties for safe subtyping.

Optimal subtyping also strongly preserves liveness properties:

Theorem 4 *Let O, U be LTS' such that* $O \sqsubseteq_{ost} U$ *and let P be an LTS describing a liveness property. Then*

$$P \sqsubseteq_{\mathcal{F}} O \Rightarrow P \sqsubseteq_{\mathcal{SF}}^{\alpha(O)} U .$$

Proof: Assume (i) $failures(O) \subseteq failures(P)$ and (ii) $O \sqsubseteq_{ost} U$, i.e. $O \sqsubseteq_{\mathcal{F}} U \setminus_c N$. By the definition of $\setminus_c$, we have
$failures(U \setminus_c N) = \{(\sigma \downarrow \alpha(O), X) \mid \exists (\sigma, Y) \in failures(U) \wedge X \subseteq Y \cup N\}$.
The rest follows again by a simple application of definitions:

$$\begin{array}{lll} & (\sigma, Y) \in failures(U) & \\ & \Downarrow & \text{Definition } \setminus_c \\ \forall X \subseteq Y \cup N: & (\sigma \downarrow \alpha(O), X) \in failures(U \setminus_c N) & \\ & \Downarrow & O \sqsubseteq_{ost} U \\ \forall X \subseteq Y \cup N: & (\sigma \downarrow \alpha(O), X) \in failures(O) & \\ & \Downarrow & P \sqsubseteq_{\mathcal{F}} O \\ \forall X \subseteq Y \cup N: & (\sigma \downarrow \alpha(O), X) \in failures(P) & \\ & \Downarrow & \text{for } X = Y \\ & (\sigma \downarrow \alpha(O), Y) \in failures(P) & \end{array}$$

and furthermore $Y \cap \alpha(O) = (Y \cup N) \cap \alpha(O)$ holds. □

The table in Figure 6 summarises the results on property preservation for the three subtyping relations.
Coming back to our example: since the second extension of the till is a safe subtype, it strongly preserves safety and weakly preserves liveness properties of the simple till. It does not strongly preserve liveness, since we now have traces after which all of the old methods are blocked. However, as long as the

	Safety		*Liveness*	
	weak	strong	weak	strong
weak subtyping	√		√	
safe subtyping	√	√	√	
optimal subtyping	√	√	√	√

Figure 6 Inheritance of properties

old features are used, the same liveness is ensured. Since a client of a till is free to choose himself what to do and since no-one else may simultaneously use the till, this is sufficient for a correct till.

5.2. CHARACTERISATION

Last we address the characterisation of subtyping relations by the properties they preserve from supertypes. Since the definitions of restriction/concealment and weak/strong preservation have already been quite close, tight connections are to be expected.

Theorem 5 *Let O, U be LTS' such that $N = \alpha(U) \setminus \alpha(O)$. Then*

- $O \sqsubseteq^{N}_{wst} U$ *if and only if* $O \sqsubseteq^{\alpha(O)}_{w\mathcal{T}} U$ *and* $O \sqsubseteq^{\alpha(O)}_{w\mathcal{F}} U$,
- $O \sqsubseteq^{N}_{sst} U$ *if and only if* $O \sqsubseteq^{\alpha(O)}_{s\mathcal{T}} U$ *and* $O \sqsubseteq^{\alpha(O)}_{w\mathcal{F}} U$,
- $O \sqsubseteq^{N}_{ost} U$ *if and only if* $O \sqsubseteq^{\alpha(O)}_{s\mathcal{T}} U$ *and* $O \sqsubseteq^{\alpha(O)}_{s\mathcal{F}} U$.

Proof: The direction "⇒" in all cases follows directly from the theorems about property preservation (by choosing $P = O$).
Direction "⇐": There are four implications to be proven here.

$O \sqsubseteq^{\alpha(O)}_{w\mathcal{T}} U \Rightarrow O \sqsubseteq_{\mathcal{T}} U \setminus_r N$.
By definition of restriction $traces(U \setminus_r N) = traces(U) \cap \alpha(O)^*$ and by weak preservation of safety $traces(U) \cap \alpha(O)^* \subseteq traces(O) \cap alpha(O)^* = traces(O)$.

$O \sqsubseteq^{\alpha(O)}_{s\mathcal{T}} U \Rightarrow O \sqsubseteq_{\mathcal{T}} U \setminus_c N$.
By definition of concealment $traces(U \setminus_c N) = traces(U) \downarrow \alpha(O)$ and by strong preservation of safety $traces(U) \downarrow \alpha(O) \subseteq traces(O) \downarrow \alpha(O) = traces(O)$.

$O \sqsubseteq^{\alpha(O)}_{w\mathcal{F}} U \Rightarrow O \sqsubseteq_{\mathcal{F}} U \setminus_r N$.
Assume $(\sigma, Y) \in failures(U \setminus_r N)$, that is, by definition of restriction

$\sigma \in \alpha(O)^*$ and there exists some $(\sigma, X) \in \mathit{failures}(U)$ such that $Y \subseteq X \cup N$. By weak preservation of liveness we know that there is also some $(\sigma, Y') \in \mathit{failures}(O)$ such that $X \cap \alpha(O) = Y' \cap \alpha(O)$. By subset closure of refusal sets it suffices to show that $Y \subseteq Y'$. Without loss of generality we assume that $\Sigma \setminus \alpha(O) \subseteq Y'$ (supertype O refuses all events outside its own alphabet anyway and we assume Y' to be maximal with respect to those events). Then $Y \subseteq Y'$ holds since all events in $\Sigma \setminus \alpha(O)$ are in Y' anyway and events from $\alpha(O)$ which are in Y are also in X and therefore in Y'.

$O \sqsubseteq_{\mathcal{SF}}^{\alpha(O)} U \Rightarrow O \sqsubseteq_{\mathcal{F}} U \setminus_c N$.

Assume $(\rho, Y) \in \mathit{failures}(U \setminus_c N)$, i.e. there is some $(\sigma, X) \in \mathit{failures}(U)$ such that $\rho = \sigma \downarrow \alpha(O)$ and $Y \subseteq X \cup N$. By strong preservation of liveness, we know that there is also some $(\sigma, Y') \in \mathit{failures}(O)$ such that $X \cap \alpha(O) = Y' \cap \alpha(O)$. We the same argument as in the last item it can be shown that $Y \subseteq Y$ holds which completes the proof. □

6. DISCUSSION

In this paper, we have investigated the preservation of properties under behavioural subtyping. We have considered both safety and liveness properties and defined weak and strong preservation with respect to methods of the superclass. At first sight, it seems that the overhead of defining preservation with respect to some set of methods is due to the formalism used, CSP. However, the same issues also arise when using for instance temporal logic for the specification of safety and liveness properties. The superclass may trivially satisfy some formula which talks about methods not present in the superclass, and the subclass may then fail to satisfy the same formula.

The results can now be applied in an incremental design with inheritance: check what kind of subtype a subclass is and then deduce what properties of the superclass are preserved. We aim at developing *syntactic conditions* on subclasses which help in checking subtyping relationships.

Furthermore we have shown that the type of preserved properties can be seen as another characterisation of the subtyping relations (beside the definitions and testing characterisations), and may additionally aid in finding the most appropriate behavioural subtyping relation for the application under consideration.

Related Work. A lot of proposed subtyping relations are based on failures and traces of classes [6, 14, 4, 3, 2]. They can be distinguished by the degree of substitutability obtained. The relation most often used is *extension*, originally defined as one correctness relation for LOTOS [5]. On the one hand extension is different from our subtyping relation since its underlying concept is *not* re-

finement, rather it allows more traces (even on old methods) in the subtype. On the other hand, extension, like weak subtyping, is also only sufficient for a single client. For none of the above cited relations the preservation of properties under subtyping is discussed.

The only work we are aware of which discusses behavioural subtyping and verification, however, in a somehow different area, is the work of Leavens and Weihl [11]. They view classes as defining *abstract data types* and consequently behavioural subtyping compares the methods of sub- and supertype with respect to their pre- and postconditions. This is a state-based view as opposed to the behavioural view on classes which we have taken here. Verification in their work is concerned with showing the correctness of implementations with respect to abstract specifications of operations via pre- and postconditions. For object-oriented programs they propose a verification technique called *supertype abstraction*: prove the soundness of an implementation with respect to the supertype specification and conclude the soundness also for all subtypes. Similar to our approach this allows to omit the re-verification for every new subtype which is created.

Acknowledgments

Thanks to Clemens Fischer for reading and commenting on an earlier version of this paper.

References

[1] 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.

[2] 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.

[3] 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.

[4] H. Bowman and J. Derrick. A junction between state based and behavioural specification. In P. Ciancarini, F. Fantechi, and R. Gorrieri, editors, *Formal methods for open object-based distributed systems FMOODS '99*, pages 213 – 239. Kluwer, 1999.

[5] E. Brinksma, G. Scollo, and Ch. Steenbergen. LOTOS specifications, their implementations and their tests. In B. Sarikaya and G. v.Bochmann, edi-

tors, *Protocol Specification, Testing and Verification VI*, pages 349 – 358. Elsevier, 1987.

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

[7] Krishna Kishore Dhara and Gary 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. A corrected version is Iowa State University, Dept. of Computer Science TR #95-20c.

[8] 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*. Springer, 2000. to appear.

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

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

[11] G.T. Leavens and W.E. Weihl. Specification and verification of object-oriented programs using supertype abstraction. *Acta Informatica*, 32:705–778, 1995.

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

[13] B. Meyer. *Object-Oriented Software Construction*. ISE, 2. edition, 1997.

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

[15] Object Management Group. *OMG Unified Modeling Language Specification*, June 1999. version 1.3.

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

[17] 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.

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

[19] 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.