Relaxing Behavioural Inheritance

Nuno Amálio
University of Luxembourg
6, rue Richard Coudenhove-Kalergi, L-1359, Luxembourg
nuno.amalio@uni.lu

Object-oriented (OO) inheritance allows the definition of families of classes in a hierarchical way. In behavioural inheritance, a strong version, it should be possible to substitute an object of a subclass for an object of its superclass without any observable effect on the system. Behavioural inheritance is related to formal refinement, but, as observed in the literature, the refinement constraints are too restrictive, ruling out many useful OO subclassings. This paper studies behavioural inheritance in the context of ZOO, an object-oriented style for Z. To overcome refinement's restrictions, this paper proposes relaxations to the behavioural inheritance refinement rules. The work is presented for Z, but the results are applicable to any OO language that supports design-by-contract.

1 Introduction

Object-oriented (OO) designs are structured around abstractions called *classes*, which represent sets of objects with certain properties in common. OO *inheritance* [20] defines families of classes with a hierarchical structure, in which higher-level abstractions (superclasses) capture state and behavioural properties that all of its specialised abstractions (subclasses) have in common. Inheritance addresses *reuse*, an important software engineering concern; in a hierarchy, subclasses reuse the behaviour of their superclasses, and add some specialised behaviour of their own.

Inheritance hierarchies have an *is-a* semantics. A child abstraction (a subclass) is a kind of a parent abstraction. The child may have extra properties, but it has a strong conceptual link with the parent; an object of a descendant is at the same time also an object of the parent class (a parent class includes all objects that are its own direct instances plus those of its descendants).

A consequence of the *is-a* semantics is *substitutability*: a subclass object can be used whenever a superclass object is expected. Substitutability is enforced in two different ways. Most OO systems enforce substitutability by checking *interface conformity* using type-checking: the signatures of the subclass operations that specialise superclass operations must conform according to certain type rules. This guarantees that subclasses can be asked to do whatever their superclasses offer. However, a subclass may comply with the interface of its superclass, but it may go along and do something different. This problem is addressed by *behavioural inheritance* [18], a strong flavour of inheritance, which enforces substitutability by checking *behavioural conformity* using proof: not only the interfaces must conform, the behaviour must conform also. This ensures that any subclass object may replace an object of the superclass without any effect on the superclass object's observable behaviour.

As observed in Liskov and Wing's seminal paper [18], behavioural inheritance is related to *data refinement*, which is also concerned with substitutability. In [15], Hoare et al define data refinement as: "One datatype (call it concrete) is said to refine another data type [...] (call it abstract), if in all circumstances and for all purposes the concrete type can be validly used in place of the abstract one." Inheritance relations should, therefore, observe a refinement relationship between subclass (concrete) and superclass (abstract). The difference is that whereas in data refinement the refinement relation varies,

John Derrick, Eerke Boiten, Steve Reeves (Eds.):
Refinement Workshop 2013
EPTCS 115, 2013, pp. 68–83, doi:10.4204/EPTCS.115.5

in behavioural inheritance this relation always follows the same pattern: a function from subclass to superclass. Behavioural inheritance is, therefore, a specialisation of data refinement.

The drawback of data refinement models of inheritance is that they are over-restrictive. Liskov and Wing [18] mention: "the requirement we impose is very strong and raises the concern that it might rule out many useful subtype relations". This paper investigates behavioural inheritance and proposes relaxations to lift certain behavioural inheritance constraints and proof obligations. The investigation is in the context of Z [24], a formal modelling language based on typed set theory and predicate calculus, with a mature refinement theory [24, 11]. The work is part of ZOO, the OO style for Z presented in [6, 2], that is the semantic domain of the *UML+Z* framework [2, 7] and the Visual Contract Language (VCL) for graphical modelling of software designs [3, 4, 5].

This paper's main contributions are three relaxations to facilitate use of rigorous behavioural inheritance in OO design, which are a result of a careful examination of mainstream OO inheritance in a Z data refinement setting using ZOO. The paper makes another contribution: it provides a way of specifying inheritance hierarchies in Z extending what is presented in [6] and that improves previous work.

2 ZOO: A Z model of OO

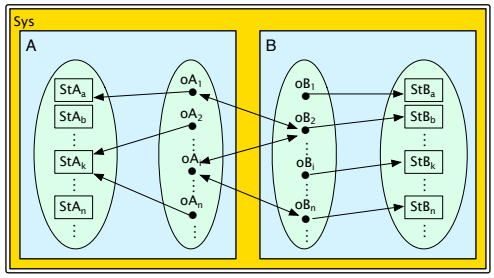


Figure 1: A ZOO system (*Sys*) made of two classes and an association between them. Class `A` comprises the set of all its object atoms (*OA*) and the set of all possible object states (*StA*); likewise for class `B`. Association comprises a set of tuples relating objects of classes `A` and `B` (sets *OA* and *OB*). A legend for the figure is given in Fig. 2c

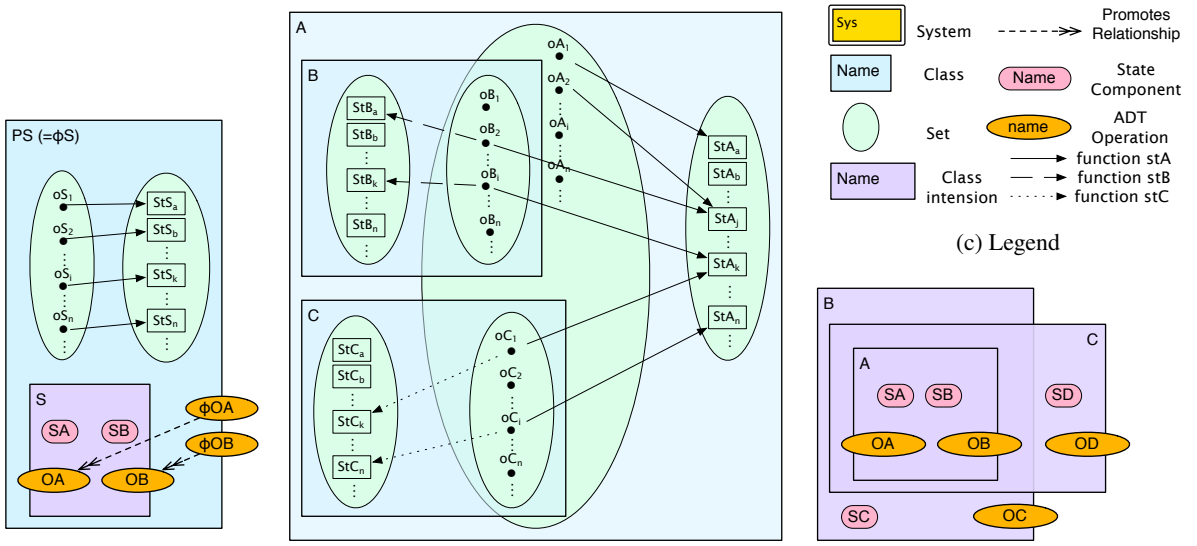
ZOO [6, 2] is an approach to specify OO models in Z: a Z style of object-orientation. Its OO model is based on Z abstract data types (ADTs) represented as Z schemas, constituting an OO model based on records [10]. ZOO is an extension of Hall's OO Z style [14, 13].

2.1 Overview

ZOO represents objects as atoms. It considers that, like a set, a class has dual meaning. *Class intension* defines a class in terms of the properties shared by its objects (for example, class `Person` with properties `name` and `address`). *Class extension* defines a class in terms of its currently existing objects (for example `Person` is $\{MrSmith, MrAnderson, MsFitzgerald\}$). This duality is expressed in terms of the representation of a class as a promoted Z ADT [24, 22] that is made up of an inner (or local) type (the class intension), and an outer (or global) type containing the actual object instances and defining the interface to the environment (the class extension). ZOO represents OO associations between classes as binary relations between the sets of objects of each class. A system is a collection of classes and their associations. Figure 1 illustrates ZOO's OO model with a OO system made of classes `A` and `B` with an association between them.

2.2 Classes and Promotion

Promotion [24, 22] is a technique to build composite structures in the Z schema calculus, so called because the inner type is *promoted* to a global state, without the need to redefine the encapsulated ADT. Typically, the state of a promoted ADT is described as a partial function $f : I \pfun S$, where I is a set of identities and S a set of states. Promotion builds operations of an ADT P in terms of operations of an



(a) A ZOO class as a Z promoted ADT

(b) The extensional viewpoint

(d) The intensional viewpoint

Figure 2: ZOO's model of inheritance. In (a), a class `PS` is represented as promoted Z ADT, separating its extensional and intensional viewpoints. Class intension (d) and extension (b) in the context of OO inheritance are illustrated with a class `A` and its subclasses `B` and `C`. In extension (b), `A` includes all its own objects (*OAs*) and all the objects of its subclasses (*OBs* and *OCs*). Each class includes a set of all possible states of its objects (*StAs*, *StBs*, *StCs*); each class has its own mapping function. In intension (d), subclasses extend superclasses with further state (state components *SC* and *SD* in Fig. 2d) and operations (*OC* and *OD*) using Z schema calculus conjunction.

encapsulated ADT S modularly (without changing S). In the context of a ZOO class, I represents a set of object identities of some class (the class extension), and S represents the state space of the class's objects (the class intention). This is depicted in Fig. 2a and captured by ZOO's class generic:

$$
\begin{array}{|l}
\hline
SCl[OS, OST] \\
os : \mathbb{P}\, OS \\
oSt : OS \pfun OST \\
\hline
\mathrm{dom}\, oSt = os \\
\hline
\end{array}
$$

Here, the parameter OS represents the set of possible objects of the class and OST represents the set of possible states of the class (the class intension).

2.3 Inheritance

A OO model of inheritance needs to consider: (a) subclassing as subsetting, (b) subclass specialisation and (c) abstract classes and polymorphism. These concerns are address by ZOO's model of inheritance, depicted in Fig.2.

- Extensionally, subclassing is subsetting. A subclass object is also an object of its superclasses; the sets of subclass objects are subsets of their superclasses. In Fig. 2b, class `A` has subclasses `B` and

C; set of A objects comprises its own (oA_i), plus those of its subclasses (oB_i and oC_i).

- Subclasses specialise or extend the state and behaviour of their superclasses. This is emphasised in the intensional viewpoint of inheritance illustrated in Fig. 2d: classes B and C extend the state and behaviour components that they inherit from A. In Fig. 2b, each subclass has its own state; the states of subclasses, however, extend the state of their superclasses. As each subclass object can been as an object of either superclass or subclass, there are mapping functions that map a subclass object to either superclass or subclass state (mapping functions in Fig. 2b). Constraints ensure that the states of subclass objects are kept consistent.

- In ZOO, abstract classes do not have direct instances. If class A in Fig. 2 were abstract, then it would just consist of objects of classes B and C. Polymorphism refers to the ability of treating objects of abstract classes polymorphically: the actual behaviour of some objects depends on their direct classes. This is a matter of selecting the right behaviour given a superclass object (the next section shows how this is specified in ZOO).

3 Specification of inheritance in ZOO

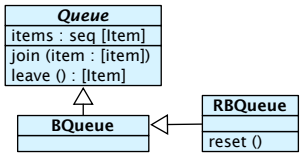


Figure 3: An inheritance hierarchy of queues formed by classes Queue, BQueue (bounded-queue), and RBQueue (resettable-bounded-queue).

ZOO's approach to inheritance is illustrated with a class hierarchy of Queues (Fig. 3). Class Queue stores a sequence of items (attribute items); it is an abstract class. It comprises two operations: join adds an element to the queue, and leave removes the element at front of the queue. Class BQueue (bounded queue) bounds the size of the queue. Class RQueue (resettable-bounded queue) introduces the extra operation reset, to empty the sequence of items.

The following builds the ZOO model corresponding to Fig. 3 for each view of the ZOO style. The full model is given at http://bit.ly/ftKZCp. The behavioural inheritance examination conducted in this paper uses this hierarchy.

3.1 Structural view

The following defines the sets *CLASS* (all class atoms), *OBJ* (all possible objects) and *abstractCl* (all abstract classes), and the relation *subCl* (subclass relation):

$$CLASS ::= QueueCl \mid BQueueCl \mid RBQueueCl$$

$$[OBJ]$$

$$abstractCl : \mathbb{P}\, CLASS$$
$$subCl : CLASS \leftrightarrow CLASS$$
$$\hline$$
$$abstractCl = \{QueueCl\}$$
$$subCl = \{BQueueCl \mapsto QueueCl, RBQueueCl \mapsto BQueueCl\}$$

In inheritance, the set of objects of a class includes its own objects and those of its subclasses. Function $\mathbb{O}$ gives all possible objects of a class. Function $\mathbb{O}_x$ gives the direct set of objects of a class (excludes objects of subclasses). These two functions are defined as:

$$\mathbb{O}_x : CLASS \to \mathbb{P}_1\, OBJ \qquad \forall\, cl, cl' : CLASS \mid cl \mapsto cl' \in subCl \bullet \mathbb{O}\, cl \subseteq \mathbb{O}\, cl'$$
$$\mathbb{O} : CLASS \to \mathbb{P}_1\, OBJ$$
$$\text{disjoint } \mathbb{O}_x$$
$$\forall\, cl : abstractCl \bullet \mathbb{O}_x\, cl = \varnothing$$
$$\forall\, cl : CLASS \bullet \mathbb{O}\, cl = \mathbb{O}_x\, cl \cup \bigcup(\mathbb{O}_x\ (\!|\ (subCl^+)^\sim\ (\!|\{cl\}|\!)\ |\!))$$

Above to the left, the first axiom says that the sets of direct objects of each class are mutually disjoint. The second says that abstract classes have an empty set of direct objects. The third defines $\mathbb{O}$ in terms of $\mathbb{O}_x$: the set of objects of a class includes its own objects and those of its descendants. Above to the right, there is a useful law that can be extracted from the axioms to the left, which says that the set of all possible objects atoms of a subclass is a subset of its superclass counter-parts.

3.2 Intensional view

3.2.1 Class `Queue`

This class comprises a sequence of items. Initially, the sequence is empty. Operation join receives an item as input and adds it to the back of the sequence. Operation leave removes and outputs the item at the head of the sequence. The intensional (or local) definition of `Queue` is as follows:

Queue[*Item*]
$$items : \text{seq}\, Item$$

QueueInit[*Item*]
$$Queue'[Item]$$
$$items' = \langle\rangle$$

QueueJoin[*Item*]
$$\Delta Queue[Item]$$
$$item? : Item$$
$$items' = items \frown \langle item? \rangle$$

QueueLeave[*Item*]
$$\Delta Queue[Item]$$
$$item! : Item$$
$$items \neq \langle\rangle$$
$$item! = head\ items \land items' = tail\ items$$

3.2.2 Class `BQueue`

The intension of BQueue is defined by extending `Queue`, its superclass. The constant *maxQ* defines the maximum number of items in the queue. The invariant states that the sequence is bound by this constant.

$$maxQ : \mathbb{N}_1$$

BQueue[*Item*]
$$Queue[Item]$$
$$\#items \leq maxQ$$

BQueueInit[*Item*]
$$BQueue'[Item]$$
$$QueueInit[Item]$$

BQueueJoin[*Item*]
$$\Delta BQueue[Item]$$
$$QueueJoin$$

BQueueLeave[*Item*]
$$\Delta BQueue[Item]$$
$$QueueLeave$$

3.2.3 Class RBQueue

Class RBQueue is defined similarly by extending BQueue. RQueue's extra operation, Reset, resets the sequence of items to the empty sequence:

$$RBQueue[Item] \;\widehat{=}\; BQueue[Item]$$

$$RBQueueInit[Item] \;\widehat{=}\; RBQueue'[Item]; BQueueInit[Item]$$

$$RBQueueJoin[Item] \;\widehat{=}\; \Delta RBQueue[Item]; BQueueJoin$$

$$RBQueueLeave[Item] \;\widehat{=}\; \Delta RBQueue[Item]; BQueueLeave$$

$$RBQueueReset[Item] \;\widehat{=}\; \Delta RBQueue[Item] \mid items' = \langle\rangle$$

3.3 Extensional View

Class extensions of all classes (abstract and non-abstract) are defined like normal classes (see [6]): by instantiating the *SCl* Z generic. State extensions of Queue, BQueue, and RBQueue are:

$$SQueue[Item] == SCl[\mathbb{O}\, QueueCl, Queue[Item]][sQueue/os, stQueue/oSt]$$

$$SBQueue[Item] == SCl[\mathbb{O}\, BQueueCl, BQueue[Item]][sBQueue/os, stBQueue/oSt]$$

$$SRBQueue[Item] == SCl[\mathbb{O}\, RBQueueCl, RBQueue[Item]][sRBQueue/os, stRBQueue/oSt]$$

For each subclassing, there is a schema expressing the required constraints, namely: (a) the set of existing objects of a subclass is a subset of its superclass, and (b) the mapping functions of both classes must be consistent. The subclassing schema for Queue/BQueue is:

$$SBQueueIsQueue[Item] \;\widehat{=}\; SQueue[Item]; SBQueue[Item] \mid sBQueue \subseteq sQueue \;\wedge\; \forall oBQueue : sBQueue \bullet (\lambda\, BQueue[Item] \bullet \theta\, Queue)(stBQueue\, oBQueue) = stQueue\, oBQueue$$

Here, the second conjunct of the predicate says that the inherited state of a BQueue object must be the same no matter the object is seen as BQueue or Queue.

Operations of non-abstract classes are formed using promotion like those of normal classes (see [6]). The update operations of BQueue, defined from the promotion frame $\Phi SBQueueUI$ defined below, are:

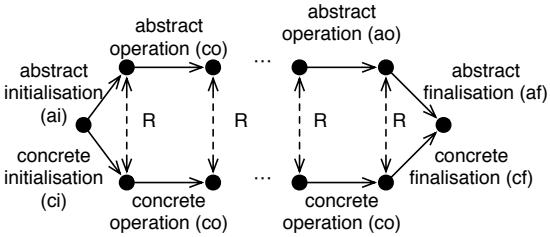
$$SBQueueJoin[Item] == \exists \Delta BQueue[Item] \bullet \Phi SBQueueUI[Item] \wedge BQueueJoin[Item]$$

$$SBQueueLeave[Item] == \exists \Delta BQueue[Item] \bullet \Phi SBQueueUI[Item] \wedge BQueueLeave[Item]$$

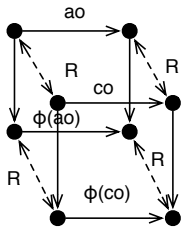
Promotion frames of subclass operations need to take the *subsetting* constraint into account. There is an intermediate frame to specify the action in the superclass, there are intermediate frames in the subclasses that extend the superclass frame. The intermediate frames for Queue and BQUeue are:

$$\Phi SQueueUI_0[Item] \;\widehat{=}\; \Delta SQueue[Item]; \Delta Queue[Item]; oQueue? : \mathbb{O}\, QueueCl \mid sQueue' = sQueue \;\wedge\; stQueue' = stQueue \oplus \{oQueue? \mapsto \theta\, Queue'\}$$

$$\Phi SBQueueUI_0[Item] \;\widehat{=}\; \Phi SQueueUI_0[Item][oBQueue?/oQueue?]; \Delta SBQueueIsQueue[Item]; \Delta BQueue[Item]; oBQueue? : \mathbb{O}\, BQueueCl \mid sBQueue' = sBQueue \;\wedge\; stBQueue' = stBQueue \oplus \{oBQueue? \mapsto \theta\, BQueue'\}$$



(a) Data refinement simulation



(b) Simulation in the context of promotion refinement

Figure 4: Simulation in the context of data refinement.

Here, the predicate of both *update* frames says that the set of existing objects remains the same, and that the state function is updated (using function overriding) for the updated object with the updated state.

The subclass final promotion frame extends the intermediate frame with the required precondition:

$\Phi SBQueueUI[Item]$

$\Phi SBQueueUI_0[Item]$

$oBQueue? \in sBQueue \cap \mathbb{O}_x\ BQueueCl$

$\theta\, BQueue = stBQueue\, oBQueue?$

Note that the promotion frames of the subclass ensure satisfaction of the subsetting constraint: whenever an object is added to the subclass it is also added to the superclass.

Polymorphic operations are specified as choice of behaviours (a disjunction). They are built in a bottom-up fashion. Polymorphic operation `BQueue.join` offers a choice between `BQueue` and `RQueue`:

$$\mathbb{S}BQueueJoin_P[Item] == \mathbb{S}BQueueJoin[Item] \lor \mathbb{S}RQueueJoin[Item]$$

The operation `join` on `Queue` offers the polymorphic operation of `BQueue`:

$$\mathbb{S}QueueJoin[Item] == \mathbb{S}BQueueJoin_P[Item]$$

3.4 Global View

The system schema includes all class extensions and subclassing schemas:

$System[Item]$

$SQueue[Item];\ SBQueue[Item];\ SRBQueue[Item]$

$\mathbb{S}BQueueIsQueue[Item] \land \mathbb{S}RBQueueIsBQueue[Item]$

4 Behavioural inheritance and Z Data Refinement

As discussed above, the correctness of inheritance hierarchies with respect to substitutability (behavioural inheritance) is checked using data refinement methods. This enables the application of the theory of data refinement, which is mature and well-developed, to the setting of OO design.

In Z, the correctness of a refinement is demonstrated using the concept of a *simulation* (Fig. 4a) with the aim of comparing ADTs inductively on a step by step basis [15]. This means that for each operation

in the abstract type, there must be a corresponding operation in the concrete type. The correctness of the refinement involves proving certain conjectures, known as simulation rules. The setting for refinement in Z is as follows: (a) find a simulation relation relating concrete and abstract data types, (b) demonstrate the correctness of the refinement by proving the required conjectures. The conjectures vary with the type of relation and the setting of refinement. A forwards (or downwards) simulation establishes a map from the concrete to the abstract type; and a backwards (or upwards) simulation is the other way round [15]. There are two settings for refinement in Z [11]: *non-blocking* (contractual) refinement interprets an operation as a contract and so outside the precondition anything may happen, whilst *blocking* (behavioural) refinement says that outside the precondition an operation is blocked.

ZOO's inheritance model allows Z data refinement to be applied to the OO setting: class refinement is simply the data refinement of the class's inner and outer ADTs. In Z, this is well studied, and known as *promotion refinement* [11, 24, 19] (Fig. 4b). One result of promotion refinement is particularly useful: under certain circumstances, promotion is compositional with respect to refinement [19]. That is, a promoted ADT refines another if there is a refinement between the types being promoted. Formally, suppose promoted ADTs *PC* and *PA* promote, respectively, *C* and *A*; then to prove that *PC* refines *PA* it is sufficient to show that *C* refines *A*. This applies when the promotion is *free* (discussed below).

Next sections study behavioural inheritance in the context of the inner type (or class intension). Behavioural inheritance conformance is checked by proving the correctness of some refinement.

4.1 A Refinement Relation for Behavioural Inheritance

To define a particular subclassing at the local level (inner type or intension), the subclass schema extends its superclass using Z schema conjunction. Formally, for a class A (abstract) and its subclass C (concrete), the state of C is defined in the intensional view by the following Z schema calculus formula, where X represents the extra state of C. In this setting, we can describe the relation between a subclass and its superclass as the function f:

$$C == A \land X \qquad\qquad f = \lambda\, C \bullet \theta A$$

This total function projects the subclass state in terms of the superclass state, removing the state added in the subclass (referred to as subclass-extra state).

4.2 Refinement rules for behavioural inheritance

Refinement rules for the above refinement function are derived in [2]. For backward and forward simulation, the rules reduce to a single set (unlike the general case, where the simulations have separate rule sets). However, as expected, some simulation rules differ for blocking and non-blocking refinements.

Let A and C be class intensions defined in Z such that C extends A (i.e. $C = A \land X$). Let A and C have initialisation schemas AI and CI, operations AO and CO, and finalisation schemas AF and CF[1]. For non-blocking (NB) refinement, C conforms to the behaviour of A, ($C \sqsupseteq A$), if and only if:

1. $\vdash ? \forall C' \bullet CI \Rightarrow AI$ (Initialisation)
2. $\vdash ? \forall C;\ i? : I \bullet \mathbf{pre}\, AO \Rightarrow \mathbf{pre}\, CO$ (Applicability)
3. $\vdash ? \forall C';\ C;\ i? : I;\ o! : O \bullet \mathbf{pre}\, AO \land CO \Rightarrow AO$ (NB Correctness)
4. $\vdash ? \forall C \bullet CF \Rightarrow AF$ (Finalisation)

[1]The finalisation condition describes a condition for the deletion of objects; e.g. a bank account may be deleted provided its balance is 0.

In a OO setting, A above corresponds to a superclass and C to a subclass. The first rule allows subclass initialisations to be strengthened. The second rule allows the precondition of a subclass operation (CO) to be weakened. The third rule says that the subclass operation must conform to the behaviour of the superclass operation (AO) whenever the superclass operation is applicable; this means that the postcondition may be strengthened. The last rule allows the finalisation to be strengthened; if the finalisation is total (the ADTs do not have a finalisation condition) the fourth rule reduces to *true*.

In the blocking setting, the correctness rule is strengthened to require the precondition to remain the same:

3a. $\vdash ? \forall C'; C; i? : I; o! : O \bullet CO \Rightarrow AO$ (B Correctness)

These rules dictate the proofs necessary for subclass initialisation, finalisation and operation specialisations (that is, operations that exist in the superclass), but not subclass-extra operations.

4.3 Extra operations

Data refinement simulation requires that for each valid execution in the concrete type there is a corresponding execution in the abstract. Each execution step in the concrete type must be simulated by the abstract. Thus, when a new operation is added to a subclass, the refinement proof needs to show that the new operation (concrete) simulates something in the superclass (abstract).

A common approach to ensure that refinement holds is to routinely include in the abstract model an operation that does nothing (called a *stuttering* step or a `skip` operation), and then prove that the new concrete operation refines `skip`. The intuition is simple. Consider an ADT as a machine operated by buttons; the user presses a button to execute an operation. In the abstract type, the `skip` operation button exists but does nothing; in the concrete type, the button executes the new operation.

The rules for checking subclass-extra operations are obtained from the rules above by replacing AO with `skip` (ΞA in Z).This imposes a constraint that the state is not changed by the operation.

4.4 Liskov and Wing [18] revisited

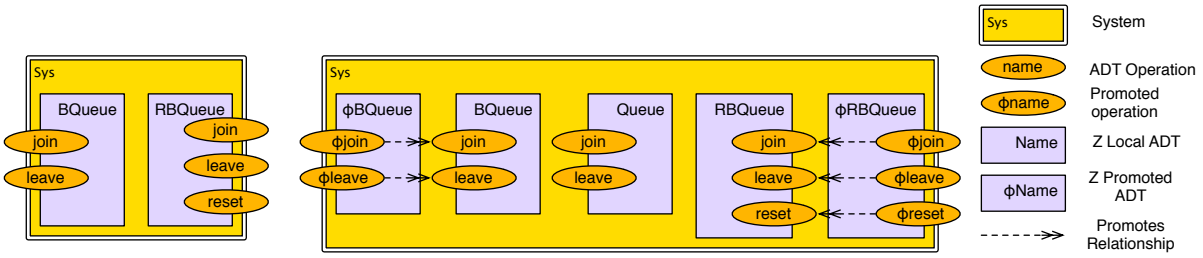
The rules above are consistent with those of Liskov and Wing [18]. They allow any function between subtype and supertype as subtyping is not necessarily inheritance; here there is only one function to reflect the specific inheritance setting. Their rules correspond to the blocking rules presented above without initialisation and finalisation. There are similar rules for extra operations, which are either a combination of those in the superclass or change subclass-extra state only (like `skip`).

5 The refinement straight-jacket

The refinement rules of the previous section are over-restrictive. Common inheritance relations of OO design, such as the queues example of section 3, cannot be deemed behavioural inheritance conformant.

5.1 Subclass-extra constraints

In the inheritance hierarchy of Fig. 3, class `BQueue` does not refine `Queue`. The applicability conjecture for the operation `join` does not hold. The precondition of `Queue.join` is, *true*, whilst that of `BQueue.join` is $\#items < maxQ$. The former does not imply the latter and so applicability fails. In refinement, the concrete type may weaken the precondition; here, the subclass precondition is stronger.



(a) Classical Refinement (b) Setting of OO Refinement used in this paper

Figure 5: Classical setting of data refinement vs OO-based data refinement used here illustrated with the queues example. Classical setting (a) assumes that operations of ADTs are exposed to the environment. In this paper's OO setting (b), it is the promoted operations of a promoted ADT (a class) that are visible to the environment; not all operations are necessarily promoted and, hence, not all of them are visible to the environment.

That the refinement proof fails is reasonable. An operation is a contract to the outside world. Abstract operation `join` contracts to do its job under any circumstance. The concrete operation, however, introduces a pre-condition. This violates substitutability, because the behaviour is observably different when the concrete type is used in place of the abstract one. Imagine a braking system of a car, where the abstract type says "upon brake slow down in any circumstance" (precondition *true*), and the concrete type says, "upon brake slow down only when speed is less than 160 Km per hour" (precondition $speed < 160$); the concrete type is obviously not a valid substitute of its abstract counter-part.

5.2 Subclass-extra operations changing inherited state

The inheritance refinement proof for `RBQueue` also fails. The `reset` operation does not refine `skip`: the correctness conjecture is not provable because `reset` violates the $\Xi Queue$ constraint by updating the inherited attribute *items*.

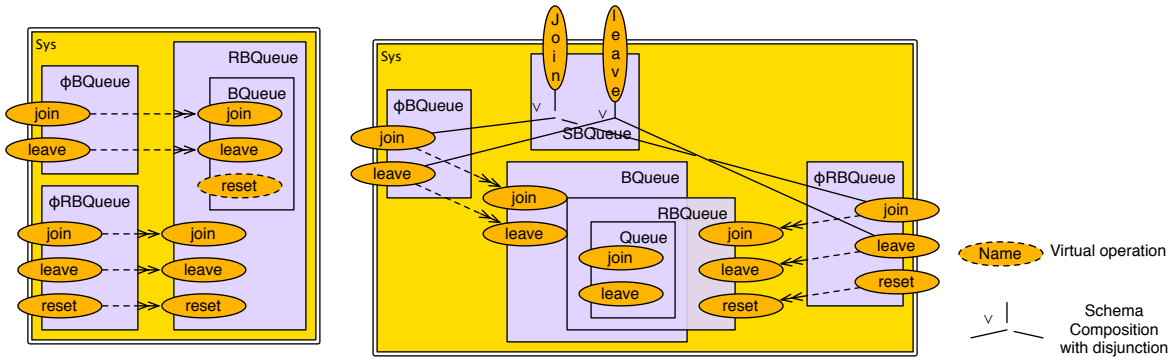
5.3 Working round the restrictions of refinement

There are two ways to address the problems imposed by the refinement restrictions: (a) we can refactor the OO models to conform to the refinement requirements, or (b) we can relax the formal restrictions.

Refactoring, a common software engineering practice, seeks to change a model whilst preserving its meaning. In this particular example, however, the refactoring would involve merging the behaviours of the three classes into a single class. The reset operation changes abstract state, so it needs moving in to the superclass. BQueue's specialised behaviour would also need to be moved to the superclass. This is a valid refactoring, albeit cumbersome: we lose the flexibility and modularity that inheritance provides.

6 Relaxations to the refinement constraints

The relaxations presented here exploit the specificities of the OO-based data refinement setting (Fig. 5). The general theory of data refinement (the classical setting) assumes that ADTs are exposed to the system environment (Fig. 5a): operations of ADTs (buttons in our metaphor) are used by the environment to interact with the system. However, ADTs are often concealed from the environment and only used internally and this applies to the OO setting based on Z promotion used here (Fig. 5b): the inner ADT



(a) Extra sub-class operations (b) OO abstract classes

Figure 6: Behavioural inheritance relaxations. In relaxation for extra sub-class operations (a), extra operation is simulated by an internal virtual operation that is never promoted and not called from the environment. In relaxation based on OO abstract classes (b), the inner ADT of the abstract superclass is never called from the environment so we can lift the applicability constraint. The operations of the abstract superclass that are made available to the environment are polymorphic (a disjunction).

(class intension) is concealed; some operations of inner ADTs may not be promoted and are, hence, also concealed. The following relaxations to behavioural inheritance exploit this.

6.1 Relaxing with virtual superclass operations

As discussed in section 4.3, the `skip` approach preserves substitutability to the environment: if pressed on the abstract type it does nothing; if pressed on the concrete type it does something, but respects the behaviour set by the abstract type. Because of restrictions of simulation in data refinement, we cannot just eliminate `skip`. What we need is to find a more *liberal* replacement. This can be found by exploiting the following: in the setting of OO inheritance a `skip`-like button will never be pressed because it is not made available to the environment (Fig. 6a). For example, when `RBQueue` is used when a `BQueue` is expected, all is needed are operations `join` and `leave`: the simulating substitute of operation `reset` is never called from the environment (Fig. 6a).

The relaxation requires a superclass *virtual* operation to simulate the operation in the subclass. In general, given a subclass-extra operation *co* there is a virtual superclass operation *ao* that simulates its behaviour. The refinement function enables the calculation of this virtual operation. Briefly, given a subclass operation (concrete):

$$co = \{CO \bullet \theta C \mapsto \theta C'\}$$

the required superclass virtual operation (abstract) is given by the formula:

$$ao = f^{\sim} \fatsemi co \fatsemi f$$

In [2], it is proved that any concrete operation (*co*) refines the calculated abstract operation (*ao*). This means that subclass extra operations can be added freely: for any concrete operation *co* there is always an abstract operation *ao* that simulates it!

Using this relaxation, `RBQueue` becomes behavioural inheritance conformant with `BQueue`. `BQueue`'s extension provides only two operations to the environment, `join` and `leave`. Internally, `RBQueue` provides those operations and `reset`; the calculated virtual operation simulates `reset` in `BQueue`'s intension and is not made available to the environment (Fig. 6a).

6.2 Relaxing by using OO abstract classes

A second relaxation exploits OO *abstract* classes[2]. In OO, an abstract class has no direct instances. Its operations definitions provide a basis for reuse and polymorphism, but are never called from the environment; they are inherently *virtual*. This is the basis of the relaxation.

The relaxation is: *if a subclass inherits from an OO abstract class, the applicability proof obligation is lifted.* The operation of an abstract class binds behaviour for its descendants, so we need to prove correctness, but there is no need to prove applicability because OO-abstract superclasses are never called from the environment. In the button analogy, when the superclass is OO-abstract, only its non-abstract subclasses can offer buttons to the environment; the operations of the abstract class are inherently virtual (Fig. 6b). This relaxation makes `BQueue` behavioural-inheritance conformant with `Queue`.

Care is needed when using OO abstract classes for relaxation: the precondition of the operation definition of an abstract class should not be relied upon to set an applicability behaviour for its descendants.

7 The effect of global constraints

The previous section demonstrated the behavioural inheritance conformance of the Queues hierarchy (Fig. 3) using the proposed relaxations. The analysis was done in a local scope, where individual objects are isolated from other objects of the system. This section analyses behavioural inheritance under a more global perspective to investigate the interference of global constraints.

7.1 Promotion Refinement Revisited

As mentioned in section 4, promotion is compositional with respect to refinement provided it is *free*. In the OO context exploited here, this means that, provided the promotions are free, checking behavioural conformance at the local level is sufficient to conclude conformance for the whole system.

A Z promotion is free if the inner type is not constrained from the global space. The following analyses the *freeness* constraint for the queues example, before showing how the freeness constraint can be relaxed so that the compositionality result is applicable to a wider range of situations.

7.2 When inner behavioural conformance is not sufficient

Consider the classes `BQueue` and `RBQueue` of figure 3. Suppose, we introduce a global constraint that affects the state of the objects of `RBQueue`, namely, the size of the sequence must be strictly less than 2:

$$\mathbb{S}RBQueue[Item]$$
$$SCL[\mathbb{O}\, RBQueueCl, RBQueue][sRBQueue/os, stRBQueue/oSt]$$
$$\forall o : sRBQueue \bullet \#(stRBQueue\, o).items < 2$$

The new constraint violates behavioural inheritance, because, under certain circumstances, the behaviour of `RBQueue` objects diverge from `BQueue` objects. Suppose that we create objects oQ of class

[2] *not* to be confused with a class that is abstract in the context of formal refinement!

BQueue and *oRQ* of class RBQueue (both queues are empty). If we execute operation join twice on them, the observed behaviour is the same. However, a third call to join on *oQ* allows the object to be added to the sequence, but fails on *oRQ* because the call is outside the precondition (there are already two items in the queue). In the non-blocking interpretation, any outcome is permitted, whilst in the blocking interpretation, the operation blocks. Substitutability is violated: *oRQ* cannot be used in place of *oQ*.

7.3 Relaxing the freeness rule

The relaxation to the freeness rule takes the form of a design guideline: *global constraints should be expressed in terms of superclasses, otherwise they may interfere with behavioural conformance proofs.* So we have relaxed from "promotion refinement is compositional provided that the inner types of the inheritance hierarchy are free from global constraints", to "promotion refinement is compositional provided global constraints affect only the inner types of classes with no ascendants".

To use this relaxation, we need to move the constraint in RBQueue to the superclass, BQueue:

$$\mathbb{S}BQueue[Item]$$
$$SCL[\mathbb{O}\,BQueueCl, BQueue][sBQueue/os, stBQueue/oSt]$$
$$\forall o : sBQueue \bullet \#(stBQueue\,o).items < 2$$

$$\mathbb{S}RBQueue[Item] == SCL[\mathbb{O}\,RBQueueCl, RBQueue][sRBQueue/os, stRBQueue/oSt]$$

Now there is no divergence: a superclass extension includes all objects of its subclasses; all subclass objects are equally affected by the constraint. If the precondition on a superclass object fails, it also fails on the objects of its subclasses.

8 Discussion

Behavioural inheritance relaxations. This paper showed how over-restrictive traditional refinement constraints can be to inheritance: many intuitive specialisations are not behavioural inheritance refinements in the strict sense. It highlights the importance of relaxations to the refinement rules: without them it is very difficult (if not impossible) to reconcile behavioural inheritance with the flexible scheme of incremental definition that makes the OO paradigm and OO inheritance so popular.

This paper proposes three relaxations to facilitate behavioural inheritance conformance:

- The first allows the addition of extra operations to the subclass freely. In [2], it is proved that for any subclass-extra operation it is possible to find a virtual operation that simulates it and satisfies the refinement constraints. This paper argues that there is no harm in introducing such operations because they are never executed; this is a property of OO systems that we can rely on. This relaxation is further confirmed by recent relaxations in other data refinement settings (see below).
- The OO abstract class relaxation is perhaps more controversial, and needs to be applied with care to avoid misunderstandings because it introduces a new kind of refinement contract that differs from the classical one. An OO abstract class operation defines a more *liberal contract*, which effectively binds a behaviour, but allows subclasses to narrow the precondition. This seems odd because it appears to allow divergent behaviour, but there is no real divergence because an OO abstract class has no direct instances. The objects of an abstract class are the instances of its subclasses only; its operations are never executed (they are virtual); collectively, the objects of an OO abstract class are *polymorphic*: they are allowed to have a multitude of behaviours that can slightly diverge from each other. As the Queue model and the other models in [2] show, with due caution this relaxation

is extremely useful; it is key in enabling OO inheritance designs that are flexible, make use of polymorphism and preserve semantic behaviour.

- The third relaxation is the result of studying how global constraints interfere with local properties. Proving behavioural conformance at the local level is not the end of the story. The assurance that the local behavioural conformance property holds in the global system rests on the compositionality of promotion with respect to refinement when the promotion is free [19]. This paper's third relaxation, a design guideline, widens the applicability of the freeness result; the paper argues informally its safety. When this relaxation is not applicable there is not a practical way to demonstrate behavioural correctness; refinement proofs of global states are very complicated even in small systems. This global relaxation has a different nature from the local relaxations given above. Whereas the local relaxations lift certain refinement constraints to allow more refinements, this global one relaxes the proof obligations, extending the freeness rule to more situations to allow more refinement definitions without the need for global proofs.

The Queues example presented here illustrates how often inheritance hierarchies are incorrectly assumed to be refinements. The subclassing of an unbounded queue by bounded one is a kind of inheritance common in the OO literature that is not, however, a behavioural inheritance refinement. In general, a bounded data type does not refine an unbounded one. This paper shows that it is possible to demonstrate behavioural conformance for the Queues inheritance model using the relaxations and without refactoring the hierarchy; we note, however, that this is not always possible. In many cases, the best solution would be to refactor the hierarchy ([2] gives some examples). All behavioural inheritance refinement proofs of the Queues example were automatically proved in Z/Eves. Usually, proofs at the level of inner (or local) types are trivial; most of them are automatically provable in Z/Eves.

Behavioural inheritance related concepts. This work helps to clarify the relation between various concepts that have distinct designations in the literature, such as, behavioural subtyping, behavioural inheritance, data refinement, class refinement and promotion refinement. The original concept of behavioural subtyping equates to data refinement in the OO setting, where an arbitrary refinement relation is allowed. Class refinement extends the theory of data refinement (which applies to ADTs) to classes; in this work class refinement equates to Z promotion refinement. Behavioural inheritance is just one specific class refinement because the refinement relation is fixed (there may be alternative formulations of this refinement relation).

ZOO and other OO models. ZOO's high-order OO model with a representation of classes as Z promotions is akin to mathematical models of OO programming languages with a formal semantics. Meyer [20] argues that the OO approach is based on the mathematical theory of abstract data types; he sees classes as having a type view and a module view, which correspond to ZOO's class intension and extension. This means that the results presented here are applicable to other OO settings with a formal semantics, especially those based on design-by-contract, such as Eiffel and JML. ZOO's model, however, differs from first-order models, such as Alloy's [17], where class fields or attributes are represented as relations; this results in models where everything is global and flat.

Multiple Inheritance. ZOO's style presented here support multiple-inheritance. [2] gives a queues example with multiple-inheritance. In the setting of multiple-inheritance, a subclass must be a behavioural-inheritance refinement of all its direct superclasses.

9 Related Work

It has long been observed the mismatch between the constraints of formal refinement and the needs of more practical software development [8]. Retrenchment [8, 9] is a more liberal approach to formal-

refinement that tries to address this problem. This paper uses this liberalisation idea in the context of mainstream OO inheritance: by studying OO inheritance in the context of data refinement, the paper is able to provide relaxations to the refinement constraints that do not violate the key substitutability principle of both refinement and behavioural inheritance.

In [1], Abrial proposes keep operations (or actions) to overcome the restrictions of the skip approach. A keep is a non-deterministic operation that is guaranteed to preserve the invariant. Abrial argues that it is safe to add keep operations to abstract types. This is similar to the superclass virtual operations proposed here, which are safe because they are not visible to the environment.

Whilst the OO model of Liskov and Wing [18] is similar to ZOO's (there is a mapping from objects (atoms) to their state), their approach is based on a earlier method of data refinement [16] that does not consider initialisation and finalisation. This paper uses data refinement based on simulation [15], the enduring basis of the theory, which accounts for object creation (initialisation) and deletion (finalisation); behavioural conformance cannot be guaranteed if these are not checked as behaviour of subclass and superclass objects may diverge. Liskov and Wing's rules correspond to the rule for blocking refinement presented here. Relaxations are not considered.

ZOO's OO inheritance approach presented here improves Hall's [14]. ZOO represents clearly a class modularly as a promoted ADT and introduces an approach to specify polymorphic operations. ZOO borrows Hall's behavioural inheritance refinement function; Hall proposes some behavioural inheritance proof rules without a formal proof in the data-refinement setting. Relaxations are not considered.

Wehrheim and Fischer [12, 23] investigate behavioural subtyping in the context of concurrency and the CSP process algebra. They studied how extra subclass operations may interfere with the behaviour of the superclass as observed from the environment, and under which conditions are safety and liveness properties preserved by the subclasses. They propose several inheritance refinement relations; the more liberal they are, the higher the risk of interference. The one that is closer to ZOO's relaxation on extra operations is *weak subtyping*, which says that the subclass should have the same behaviour as its superclass as long as no extra operations are called; the extra operations are not considered in the comparison. The authors also proposed a more restricted relation, *optimal subtyping*, which does not allow altering the behaviour of the superclass at all; it is the same as the skip behaviour.

Object-Z [21, 11] defines a formal semantics for inheritance and a notion of class refinement, but a discussion of behavioural inheritance is generally absent in its books. In [11], behavioural inheritance and its relation to refinement is discussed, but no proof obligations are proposed to check its correctness.

10 Conclusions

This paper investigates behavioural inheritance using ZOO, a Z style of object-orientation, and the theory of Z data refinement. It shows how over-restrictive refinement constraints are to inheritance, and how important it is to relax such constraints in order to reconcile incremental definition with behavioural conformance. The paper proposes two new relaxations to the refinement rules that do not compromise the principles of data refinement allowing more refinements than the classical setting. The paper also shows how global properties can interfere with behavioural inheritance conformance that is only proved locally, and proposes a relaxation to the proof obligations at the global level that allows the important property of composition of promotion with respect to refinement to be more widely applicable in a OO setting, allowing more refinement definitions without the need for global proofs. This paper's main contributions are these three relaxations addressing behavioural inheritance conformance, which are the result of a careful examination of OO inheritance in the setting of data refinement. To the author's knowledge, these relaxations have not been proposed before. A secondary contribution is the approach to specify inheritance in Z that improves Hall's approach [14].

Acknowledgements. Many thanks to Susan Stepney and Fiona Polack for their helpful comments, insight and encouragement on this work.

References

[1] J.R. Abrial, D. Cansell & D. Méry (2005): *Refinement and Reachability in Event_B*. In: *ZB2005*, *LNCS* 3455, Springer, pp. 222–241, doi:10.1007/11415787_14.

[2] N. Amálio (2007): *Generative frameworks for rigorous model-driven development*. Ph.D. thesis, Dept. Computer Science, Univ. of York.

[3] N. Amálio, C. Glodt & P. Kelsen (2011): *Building VCL models and automatically generating Z specifications from them*. In: *FM 2011*, *LNCS* 6664, Springer, pp. 149–153, doi:10.1007/978-3-642-21437-0_13.

[4] N. Amálio & P. Kelsen (2010): *Modular Design by Contract Visually and Formally using VCL*. In: *VL/HCC 2010*, IEEE, pp. 227–234, doi:10.1109/VLHCC.2010.39.

[5] N. Amálio, P. Kelsen, Q. Ma & C. Glodt (2010): *Using VCL as an Aspect-Oriented Approach to Requirements Modelling*. *TAOSD* VII, pp. 151–199, doi:10.1007/978-3-642-16086-8_5.

[6] N. Amálio, F. Polack & S. Stepney (2005): *An Object-Oriented Structuring for Z based on Views*. In: *ZB 2005*, *LNCS* 3455, Springer, pp. 262–278, doi:10.1007/11415787_16.

[7] N. Amálio, F. Polack & S. Stepney (2006): *UML+Z: Augmenting UML with Z*. In H. Abrias & M. Frappier, editors: *Software Specification Methods*, ISTE, doi:10.1002/9780470612514.ch5.

[8] R. Banach & M. Poppleton (1998): *Retrenchment: An engineering variation on refinement*. In: *B'98*, *LNCS* 1393, Springer, pp. 129–147, doi:10.1007/BFb0053358.

[9] R. Banach, M. Poppleton, C. Jeske & S. Stepney (2007): *Engineering and theoretical underpinnings of retrenchment*. *Science of Computer Programming* 67(2–3), pp. 301–329, doi:10.1016/j.scico.2007.04.002.

[10] L. Cardelli (1988): *A semantics of multiple inheritance*. *Information and Computation* 76, pp. 138–164, doi:10.1016/0890-5401(88)90007-7.

[11] J. Derrick & E. Boiten (2001): *Refinement in Z and Object-Z: foundations and advanced applications*. Springer.

[12] C. Fischer & H. Wehrheim (2000): *Behavioural subtyping relations for object-oriented formalisms*. In: *AMAST 2000*, *LNCS* 1816, Springer, pp. 469–483, doi:10.1007/3-540-45499-3_33.

[13] A. Hall (1990): *Using Z as a Specification Calculus for Object-Oriented Systems*. In A. Hoare, D. Bjørner & H. Langmaack, editors: *VDM '90*, *LNCS* 428, Springer, pp. 290–318, doi:10.1007/3-540-52513-0_16.

[14] A. Hall (1994): *Specifying and Interpreting Class Hierarchies in Z*. In: *Z User Workshop*, Workshops in Computing, Springer, pp. 120–138, doi:10.1007/978-1-4471-3452-7_8.

[15] J. He, A. Hoare & J.W. Sanders (1986): *Data Refinement Refined*. In: *ESOP'86*, *LNCS* 213, Springer, pp. 187–196, doi:10.1007/3-540-16442-1_14.

[16] A. Hoare (1972): *Proof of Correctness of data representations*. *Acta Informatica* 1(1), pp. 271–281, doi:10.1007/BF00289507.

[17] D. Jackson (2006): *Software Abstractions: logic, lanaguage, and analysis*. MIT Press.

[18] B. Liskov & J. Wing (1994): *A Behavioral Notion of Subtyping*. *ACM Trans. Program. Lang. Syst.* 16(6), pp. 1811–1841, doi:10.1145/197320.197383.

[19] P.J. Lupton (1990): *Promoting forward simulation*. In: *Z User Workshop*, Springer, pp. 27–49.

[20] B. Meyer (1997): *Object-Oriented Software Construction*. Prentice-Hall.

[21] G.P. Smith (2000): *The Object-Z Specification Language*. Kluwer Academic Publishers, doi:10.1007/978-1-4615-5265-9.

[22] S. Stepney, F. Polack & I. Toyn (2003): *Patterns to Guide Practical Refactoring: examples targetting promotion in Z*. In: *ZB 2003*, *LNCS* 2651, Springer, pp. 20–39, doi:10.1007/3-540-44880-2_3.

[23] H. Wehrheim (2000): *Behavioral Subtyping and property preservation*. In S.F. Smith & C.L. Talcott, editors: *FMOODS 2000*, Kluwer, pp. 213–231, doi:10.1007/978-0-387-35520-7_11.

[24] J. Woodcock & J. Davies (1996): *Using Z: Specification, Refinement, and Proof*. Prentice-Hall.