Chapter 1
Sound and Relaxed Behavioural Inheritance

Nuno Amálio

Abstract Object-oriented (OO) inheritance establishes taxonomies of OO classes. Behavioural inheritance (BI), a strong version, emphasises substitutability: objects of child classes replace objects of their ascendant classes without any observable effect difference on the system. BI is related to data refinement, but refinement's constrictions rule out many useful OO subclassings. This paper revisits BI at the light of Z and the theory of data refinement. It studies existing solutions to this problem, criticises them, and proposes improved relaxations. The results are applicable to any OO language that supports design-by-contract (DbC). The paper's contributions include three novel BI relaxations supported by a mathematical model with proofs carried out in the Isabelle proof assistant, and an examination of BI in the DbC languages Eiffel, JML and Spec♯.

Key words: refinement; Z; object-orientation, inheritance, design-by-contract; JML; Eiffel; Spec♯; behavioural subtyping

1.1 Introduction

The object-oriented (OO) paradigm has a great deal in common with biological classification (taxonomy) and the ever present human endeavour to establish taxonomies that reflect degree of relationship [30]. This is, perhaps, a factor behind OO's popularity, which uses classification to tame diversity and complexity. Whilst biologists go from life into classification, computer scientists use classification as templates that generate computing life.

Nuno Amálio
School of Computing and Digital Technology, Birmingham City University, Millennium Point, Curzon Street, Birmingham B4 7XG, UK, e-mail: nuno.amalio@bcu.ac.uk

OO design builds taxonomies around *classes*: abstractions representing *living* computing objects with common characteristics. This resembles biological classifications, which group similar entities into taxa [30]. OO classes define both static and dynamic characteristics. Each object of a class is a distinct individual with its own identity. A class has a dual meaning: intension and extension [20]. Intension sees a class in terms of the properties shared by all its objects (for example, a class `Person` with properties `name` and `address`), whereas extension views a class in terms of its living objects (for example, class `Person` is {MrSilva, MsHakin, MrPatel}).

Biological classifications hierarchically relate taxa through common ancestry [30]. This is akin to OO *inheritance*, which builds hierarchies from similarity to specificity in which higher-level abstractions (*superclasses* or ancestors) capture common characteristics of all descendant abstractions (subclasses). Inheritance provides a *reuse* mechanism: descendants reuse their ancestor definitions, and may define extra characteristics of their own.

The essence of OO inheritance lies in its *is-a* semantics. A child abstraction (a subclass) is a kind of a parent abstraction. The child may have extra characteristics, but it has a strong link with the parent: a living object of a descendant is at the same time also an object of its ascendant classes, and a parent class includes all objects that are its own direct instances plus those of its descendants. For example, when we say that a human is a primate, then any person is both a human and a primate; characteristics of primates are also characteristics of humans, however humans have characteristics of their own which they do not share with other primates.

OO computing life becomes more complicated when it comes to dynamics or behaviour, which is concerned with objects doing something when stimulated through *operations*. To understand operations, we resort to a *button* metaphor: operations are buttons, part of an object's interface, triggered from the outside world to affect the internal state of an object and produce observable outputs. Often, such button-pushes require data (or inputs). Inheritance implies that ancestor buttons belong to descendants also and that descendants may specialise them. For example, `walk` on primate could be specialised differently in humans and gorillas as upright-walk and knuckle-walk, respectively, to give rise to *polymorphism*. Operations apply the principle of information hiding; we have the *interface* of an operation — its name and expected data with respect to inputs outputs —, which is what the outside world sees, and its definition in terms of what it actually does (programs at a more concrete computing level). Through operations the outside world derives and instils meaningful outcomes from and into the abstraction.

Inheritance's *is-a* semantics entails *substitutability*: a child object can be used whenever a parent object is expected. For instance, a human is suitable whenever a primate is expected. This, in turn, entails a certain uniformity to prevent unwanted divergence, which is enforced at two levels. *Interface conformity*, the more superficial level, requires that the interfaces of the shared buttons, in sub- and super-class, conform with each other with respect to the

data being interchanged (inputs and outputs)[1]. This guarantees that subclasses can be asked to do whatever their superclasses offer, but this leaves room for undesirable deviation. For example, a gorilla class may comply with a primate `walk` button, but be actually defined just as standing upright without any movement. The second deeper level of enforcement tackles this issue through *behavioural inheritance* (BI) [28, 21]: not only the interfaces must conform, the behaviour must conform also to ensure that subclass objects may stand for superclass objects without any difference on the object's observable behaviour from a superclass viewpoint. In our primates example, just standing upright would fail to meet the underlying motion expectations. Only through proof can the satisfaction of BI be verified.

Deep substitutability is captured by the theory of data refinement [22, 39, 16]. Inheritance relations induce refinement relations between parent and child[2]. This paper tackles the refinement restrictions, a major obstacle to BI's ethos of correctness already acknowledged by Liskov and Wing [28].

This paper delves into BI's foundations to propose relaxations that tackle refinement's overkills and constrictions. The investigation is in the context of Z [39, 24], a formal modelling language with a mature refinement theory [39, 16]. The work builds up on ZOO, the OO style for Z presented in [10, 2, 3], that is the semantic domain of $UML + Z$ [11, 2, 12] and the Visual Contract Language (VCL) for graphical modelling of software designs [7, 8, 9, 6].

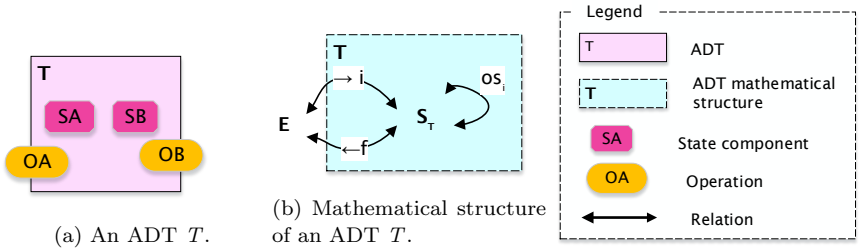
Contributions. This paper's contributions are follows:

- The paper presents four relaxations to BI. Three of these relaxations are novel. A fourth relaxation has been proposed elsewhere, but lacked a formal proof; it is proved here with the aid of the Isabelle proof assistant.
- A thorough examination of the BI relaxations that underpin the design by contract languages JML, Eiffel and Spec♯.

Paper outline. The remainder of this paper is as follows. Section 1.2 presents the mathematical model that underpins the paper's BI study. Section 1.3 introduces the paper's BI setting and derives conjectures for BI. Section 1.4 presents the running example, which is analysed in section 1.5 to better understand how BI's restrictions affect inheritance. Section 1.6 performs a thorough examination of BI in the DbC languages JML, Eiffel and Spec♯. Section 1.7 presents the paper's four relaxations which are applied to the running example. Finally, the paper concludes by discussing its results (section 1.8), comparing the results with related work (section 1.9) and by summarising its main findings (section 1.10). The accompanying technical report [4] provides supplementary material not included in the main text.

[1] This involves type-checking, a computationally efficient means of verification which checks that variables hold valid values according to their types (e.g. boolean variables cannot hold integers).

[2] Whereas in data refinement the refinement relation varies, in BI this relation is always a function from subclass to superclass. BI is a specialisation of data refinement.



(a) An ADT T.

(b) Mathematical structure of an ADT T.

Fig. 1.1: An abstract data type (ADT) (a) comprises state (SA, SB) and operations (OA, OB). Mathematically (b), an ADT T is made-up of a set of states S_T, an initialisation i (a relation from environments E to states S_T), a finalisation f (a relation from states S_T to environments E) and operations os (an indexed set of relations between states S_T).

1.2 An Abstract Mathematical Model of OO

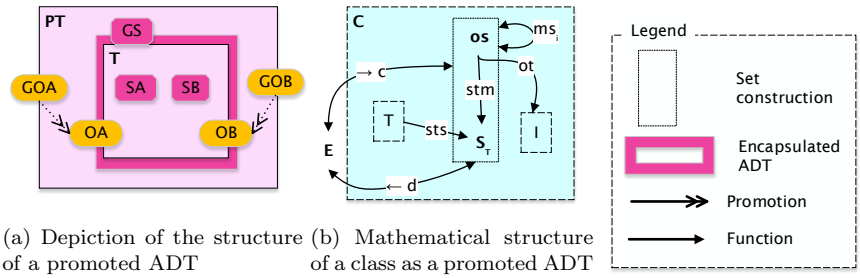
This section presents the paper's OO mathematical model, drawn from ZOO [10, 2, 3], our approach to couch OO models in Z. The model rests on abstract data types (ADTs), enabling a connection to data refinement.

The sequel refers to mathematical definitions of appendix 1.A, which abridge the definitions of the accompanying technical report [4].

1.2.1 The ADT foundation

ADTs, depicted in Fig. 1.1, are used to represent state-based abstractions made-up of structural and dynamic parts that capture computing life-forms. They comprise a state definition and a set of operations (Fig. 1.1a).

Figure. 1.1b depicts ADTs' mathematical underpinnings. There are sets of all possible type states S, all possible environments E, all possible identifiers I, and all possible objects O (def. 1). An ADT (def. 3) is a quadruple $T = (S_T, i, f, os)$ comprising a set of states $S_T \subseteq S$ (the state space), an initialisation $i : E \leftrightarrow S_T$ (a relation between environment and state space), a finalisation $f : S_T \leftrightarrow E$ (a relation between state space and environment) and an indexed set of operations $ops : I \nrightarrow S_T \leftrightarrow S_T$ (a function from operation identifiers to relations between states). Functions sts, ini, fin, and ops (def. 3) extract the different ADT components (e.g. for T above, $sts\ T = S_T$).



(a) Depiction of the structure of a promoted ADT

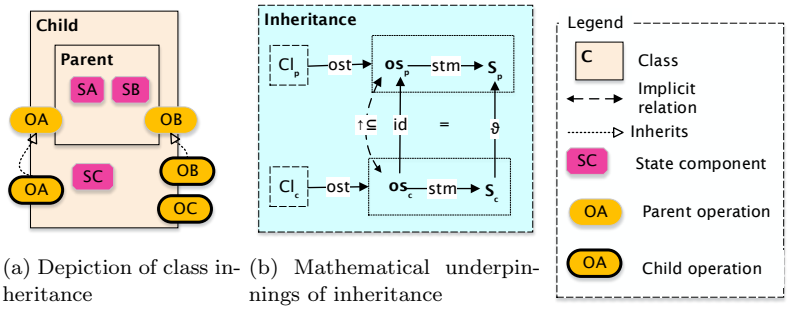
(b) Mathematical structure of a class as a promoted ADT

Fig. 1.2: Promoted abstract data types (PADTs) (a) comprise an inner type T and global operations (GOA, GOB) promoting inner operations (OA, OB); OO classes are PADTs here. Mathematically (b), classes have a global function (stm) mapping class objects os (identities) to their inner states (type T); this is the basis for constructor (c from inner initialisation), destructor (d, from inner finalisation) and modifier operations (ms from inner operations).

1.2.2 Classes

ADTs lack an intrinsic identity. In the model of Fig. 1.1b (def. 3), two ADT instances with the same state denote a single instance. Classes are populations of individuals, suggesting uniquely identifiable individuals that retain their identity irrespective of the state in which they are in. Z promotion [39, 35] is a modular technique that builds ADTs with a collective identity by promoting a local ADT in a global state without the need to redefine the encapsulated ADT; promoted operations are framed, as only a portion of the global state changes. Figure 1.2 depicts classes as *promoted* ADTs (PADTs), which underpin the OO model presented here. A PADT PT (Fig. 1.2a), is made up of an inner (or local) type T that is encapsulated and brought into a global space to make the compound (or outer) type PT. The inner type represents class intension; the outer type is concerned with class extension.

The mathematical underpinnings of a class as PADT are pictured in Fig. 1.2b. A class (def. 6) is a 9-tuple $C = (ci, t, os, stm, ot, c, d, ms)$, comprising a class identifier $ci : I$, an inner type $t : ADT$, a set of objects $os \subseteq O$ of all possible object identities of the class, a global class state made up of an object to state mapping $stm \in os \pfun sts\ t$, a typing mapping $ot \in os \pfun I$ indicating the direct class of the classes living objects, a constructor class operation $c \in nOps$, a destructor class operation $d \in dOps$ and class modifiers $ms \in uOps$ (see def. 5 for $nOps$, $dOps$ and $uOps$). Functions icl, ity, osu, los, $csts$, ost, ocl, cop, dop and $mops$ extract information from class compounds (def. 6) to yield: class identifier (icl), inner type (ity), universe of the class's possible objects (osu), class's living objects (los), set of class states ($csts$), object to state mapping (ost), object to direct class mapping (ocl), and constructor (cop), destructor (dop) and modifier operations ($mops$).



(a) Depiction of class inheritance

(b) Mathematical underpinnings of inheritance

Fig. 1.3: Class inheritance: descendants inherit their ancestor characteristics and add something of their own. In (a), state and operations of `Parent` are inherited by `Child`, which adds SC and OC. Mathematically (b), inheritance involves a pair of functions that preserve the class's object to state mappings (expressed as diagram commutativity); one function is identity (id) — child objects are a subset of their parent objects —, the other is the abstraction function ϑ, which gives the state as a parent when given the state of a child.

1.2.3 Inheritance

Inheritance, pictured in Fig. 1.3, embodies a constructive approach that builds specificity on top of commonality (Fig. 1.3a). Its is-a semantics means that a child is a parent with possibly something extra (Fig. 1.3a), which has implications at the level of inner and outer types.

The inner type captures how the child inherits the characteristics of the parent and adds something of its own through ADT extension (def. 8), expressed in Z as schema conjunction (def. 9): $C == A \wedge X$. Given inner ADTs C (concrete or child) and A (abstract or parent), then C is defined as being A with something extra, X.

From a child state space it is possible to derive the parent's (by removing what is extra) as captured in the abstraction function ϑ (def. 10). The mathematical relation between classes parent C_p and child C_c, depicted in Fig. 1.3b, rests on this ϑ function that maps inner object states of child to the ones of parent; in all states of the system the relation between child and parent must preserve the diagram commuting of Fig. 1.3b. This relation, described as a mapping from child to parent (def. 11), materialises the is-a semantics at the mathematical level of classes in terms of the commuting of Fig. 1.3b: at any system state a child object can be seen as a parent object.

Abstract classes (def. 12) have no direct instances and, hence, lack a direct existence. They capture general ancestors in a hierarchy. For example, a class primate would be an abstract class because its existence is indirectly defined by the specimens of its descendants, such as human and gorilla.

1.3 Behavioural Inheritance (BI) and Refinement

The following investigates BI under the prism of data refinement; it refers to mathematical definitions from appendix 1.A.

1.3.1 Data Refinement

Refinement is a stepwise approach to software development, in which abstract models are increasingly refined into more concrete models or programs with each step carrying certain design decisions [38]. Data refinement [22] provides a foundation to this process through a theory that compares ADTs with respect to substitutability and preservation of meaning.

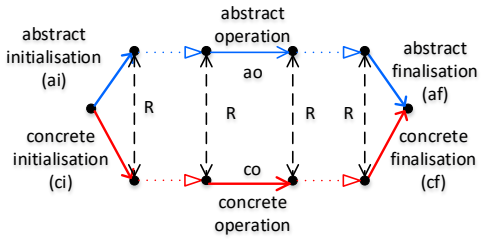


Fig. 1.4: Data refinement simulation: every step in the concrete type is simulated by a step in the abstract type.

Data refinement is founded on total relations [22]; operations are relations over a data type, programs are sequence of operations. Complete programs over a ADT start with an initialisation, carry out operations and end with a finalisation (def. 13). In this setting, data refinement is set inclusion: given ADTs C and A, then for all complete programs p_C and p_A, with the same underlying operations over C and A respectively, C refines A ($C \sqsupseteq A$) if and only if $p_C \subseteq p_A$ (def. 14).

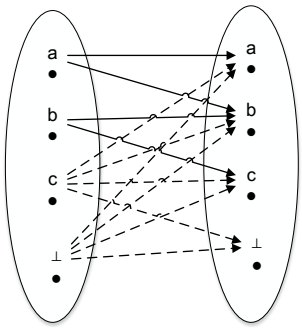


Fig. 1.5: Contractual totalisation of a relation [39].

It is difficult to prove refinements through complete programs. In practice, refinement proofs resort to *simulations* (Fig. 1.4) where ADTs are compared inductively [22] through a simulation relation (R in Fig 1.4). For each operation in the abstract type, there must be a corresponding operation in the concrete type. A refinement is verified by proving conjectures (or simulation rules), given in definitions 15 for forwards (or downwards) simulation and 16 for backwards (or upwards) simulation and which are related to the three commutings of Fig 1.4. The two types of simulations are sufficient for refinement (anything they can prove is a refinement) and together they are necessary (any refinement can be proved using either one of them) [22].

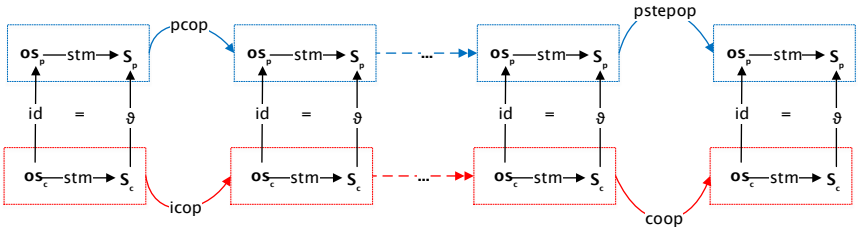


Fig. 1.6: Behavioural inheritance class simulation. An inherited class operation `icop` is simulated by a corresponding parent class operation `pcop` from which it inherits. There may be child only class operations (represented as `coop`), which are simulated by some step operation in the parent (`pstepop`).

In OO with design by contract [31], operations are described in terms of pre- and post-conditions. Operations are *partial* relations applicable only in those ADT states that satisfy the pre-condition (the relation's domain). The language Z operates in this partial setting. Refinement based on total relations is adapted to Z in [39] by deriving simulation rules based on a totalisation of partial relations. There are two Z refinement settings [16]: *non-blocking* (contractual) refinement interprets an operation as a contract and so outside the precondition anything may happen, while *blocking* (behavioural) refinement says that outside the precondition an operation is blocked. Figure 1.5 gives the contractual totalisation of relation $r = \{a \mapsto a, a \mapsto b, b \mapsto b, b \mapsto c\}$ where undefinedness ($\bot$) and all elements outside the relation's domain are mapped to every possible element in the target set augmented with $\bot$.

This paper focuses on the contractual interpretation, the most relevant for our OO context; [4] covers both interpretations. Simulation rules for contractual refinement are given in facts 1 (forwards) and 2 (backwards).

1.3.2 BI Refinement

Although developed to support refinement (from abstract to concrete) or abstraction (other way round), data refinement compares data types with respect to substitutability making it applicable to BI.

BI needs to relate the types being compared (R in Fig. 1.5). Such a relation can be discerned from Fig. 1.3b by looking into how child and parent are related through inheritance (def. 11). This gives a basis for the BI class simulation portrayed in Fig. 1.6, depicting inherited child operations (`icop`) simulated by the parent operation they inherit from (`pcop`) and child-only operations being simulated by parent step operations. Figure 1.6 suggests a refinement relation as illustrated in Fig. 1.3b made up of a morphism comprising two functions: the ϑ abstraction function (def. 10) and identity. This

hints at a modular approach for BI: we start with inner type refinement (class intension) through function ϑ, followed by outer type refinement (class extension), which is related to Z promotion refinement [16, 39, 29].

1.3.3 Inner BI as ADT Extension Refinement

For any classes $Cl_A, Cl_C : Cl$ (def. 6), such that Cl_C is a child of Cl_A (Cl_C **inh** Cl_A, def. 11), we have that inner BI equates to ADT extension refinement of the class's inner types: $ity\ Cl_C \sqsupseteq_{Ext} ity\ Cl_A$.

Two alternative extension refinement settings are considered: one based on the general function ϑ (def. 17) and a ϑ specific to the Z schema calculus (def. 18) to cater to the ZOO approach. Simulation rules were derived with the aid of Isabelle (see [4] for details). For backwards and forwards simulation, the rules reduce to a single set (unlike the general case with separate rule sets) — corollary 1 of appendix 1.A, a consequence of fact 3.

Let $A, C : ADT$ be two ADTs — such as the inner ones of classes Cl_C and Cl_A above, such that $A = ity\ Cl_A$ and $C = ity\ Cl_C$ — where C **extends** A (def. 8). If C and A are two Z schema ADTS, then their their relation is described by the schema calculus formula $C == A \wedge X$. Let A and C have initialisation schemas AI and CI, operations AO and CO, and finalisation schemas AF and CF[3]. As established by fact 5, $C \sqsupseteq_{Ext} A$ if and only if:

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

The first rule allows initialisations to be strengthened. The second rule allows the weakening of the precondition of a concrete operation (CO). The third rule says that the extended operation (CO) must conform to the behaviour of the base operation (AO) whenever the base operation is applicable — the postcondition may be strengthened. The last rule allows finalisation strengthening, but reducing to *true* if the finalisation is total (the ADTs lack a finalisation condition). Fact 4 captures extension refinement in the more general relational setting.

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

1.3.4 Extra operations

Refinement requires that each execution step in the concrete type is simulated by the abstract. A non-inherited operation in a child class (concrete) needs to simulate something in the parent (abstract).

A common approach to this issue involves an abstract operation that does nothing and changes nothing (called a *stuttering* or a `skip` operation). The proofs verify that the new concrete operation refines `skip`: in the abstract type, `skip` does nothing; in the concrete type, the button executes the new operation. The rules for checking child-extra operations are obtained from the rules above by replacing AO with `skip` (ΞA in Z).

1.3.5 Outer BI

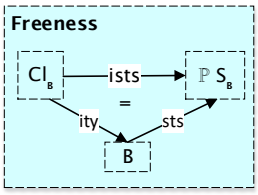


Fig. 1.7: Class (or promotion) freeness

The BI simulation rules above (section 1.3.3) cater to the class's inner (or local) ADT only. In the class's outer ADT, the concern is whether the refinement proved locally is preserved globally.

Z *promotion refinement* relies on promotion freeness [16, 39, 29]: a class refines another if there is a refinement between the inner types and the child class is free or unconstrained from the global state [29]. Figure 1.7 describes freeness as a diagram commuting: a class is free if the set of global object states (function *ists*) is the same as the set of states of its inner type (function composition $sts \circ ity$) — as per def. 19. Hence, given classes $Cl_A, Cl_C : Cl$ (def 6) such that Cl_C is a child of Cl_A (Cl_C **inh** Cl_A, def. 11), as per fact. 6, class Cl_C BI-complies with Cl_A ($Cl_C \sqsupseteq_{BI} Cl_A$) when $ity\ Cl_C \sqsupseteq_{Ext} ity\ Cl_A$ and the classes are free (Fig. 1.7, def. 1.7). Hence, the rules of section 1.3.3 can be carried safely to contexts in which freeness holds.

1.4 The ZOO model of Queues

Figure 1.8 presents the running example of a hierarchy of queues. Class `QueueManager` holds an indexed set of queues (`HasQueues`); the hierarchy is as follows:

- Abstract class `Queue` holds a sequence of `items`. It has two operations: `join` adds an element to the queue, and `leave` removes the queue's head.
- Class `BQueue` (bounded queue) bounds the size of the queue.

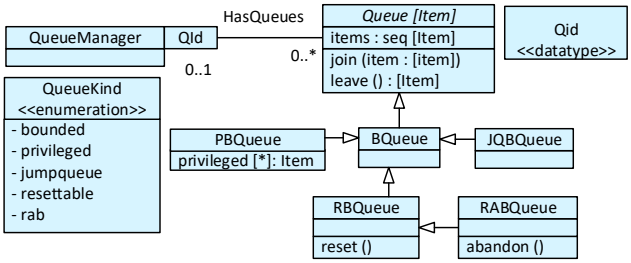


Fig. 1.8: A UML class diagram describing an inheritance hierarchy of queues made-up of classes `Queue`, `BQueue` (bounded-queue), `RBQueue` (resettable-bounded-queue), `RABQueue` (resettable-abandonable-bounded-queue).

- Class `PBQueue` (privileged-bounded queue) reserves the last place in the queue to some privileged item.
- Class `RBQueue` (resettable-bounded queue) adds operation `reset` to empty the queue.
- Class `JQBQueue` (jump-the-bounded-queue) add an extra behaviour to operation `join`: the item taking the queue's last place jumps the queue.
- Class `RABQueue` (resettable-abandonable-bounded-queue) adds `abandon` enabling any element to leave the queue irrespective of its position.

The following presents excerpts of the ZOO model formalising the class diagram of Fig. 1.8. The complete model is given in [4]. Further information on ZOO can be obtained from [4, 3, 2, 10].

1.4.1 ZOO model Excerpt: Inner ADTs

Inner ADT of class `Queue` holds a sequence of items, which is initially empty. Operation `join` receives an item and adds it to the back of the sequence. Operation `leave` removes and outputs the sequence's head.

$$\begin{array}{|l} Queue[Item] \\ \hline items : \mathrm{seq}\ Item \\ \hline \end{array}$$

$$\begin{array}{|l} QueueInit[Item] \\ \hline Queue[Item]' \\ \hline items' = \langle\rangle \\ \hline \end{array}$$

$$\begin{array}{|l} QueueJoin[Item] \\ \hline \Delta Queue[Item] \\ item? : Item \\ \hline items' = items \frown \langle item? \rangle \\ \hline \end{array}$$

$$\begin{array}{|l} QueueLeave[Item] \\ \hline \Delta Queue[Item];\ item! : Item \\ \hline items \neq \langle\rangle \land item! = head\ items \\ items' = tail\ items \\ \hline \end{array}$$

BQueue extends Queue by bounding the queue with constant maxQ.

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

BQueue[*Item*]
Queue[*Item*]

$\# items \leq maxQ$

BQueueInit[*Item*]
BQueue[*Item*]′
QueueInit[*Item*]

BQueueLeave[*Item*]
Δ*BQueue*[*Item*]
QueueLeave[*Item*]

BQueueJoin[*Item*]
Δ*BQueue*[*Item*]
QueueJoin[*Item*]

RBQueue extends BQueue; extra operation Reset empties the queue.

RBQueue[*Item*]
BQueue[*Item*]

RBQueueReset[*Item*]
Δ*RBQueue*[*Item*]

$items' = \langle\rangle$

PBQueue extends BQueue by adding a set of privileged items, set at initialisation, and reserving the last place in the sequence to such an item.

PBQueue[*Item*]
BQueue[*Item*]
$privileged : \mathbb{P}_1\ Item$

$\# items = maxQ \Rightarrow last\ items \in privileged$

PBQueueInit[*Item*]
PBQueue[*Item*]′
BQueueInit[*Item*]
$privileged? : \mathbb{P}_1\ Item$

$privileged' = privileged?$

JQBQueue slightly modifies operation join inherited from BQueue: the item occupying the last place left in the queue is placed at the queue's head.

JQBQueue[*Item*]
BQueue[*Item*]

JQBQueueInit[*Item*]
JQBQueue[*Item*]′
BQueueInit[*Item*]

JQBQueueLeave[*Item*]
Δ*JQBQueue*[*Item*]
BQueueLeave[*Item*]

JQBQueueJoin[*Item*]
Δ*PBQueue*[*Item*]; $item? : Item$

$\# items < maxQ - 1 \Rightarrow$ *BQueueJoin*[*Item*]

$\# items = maxQ - 1 \Rightarrow items' = \langle item? \rangle \frown items$

RABQueue extends RBQueue by adding abandon, allowing elements to leave the queue no matter their position.

RABQueue[*Item*]
RBQueue[*Item*]

RABQueueInit[*Item*]
RABQueue[*Item*]′
BQueueInit[*Item*]

$$\begin{array}{|l} ABQueueLeave[Item] \\ \Delta RABQueue[Item] \\ RBQueueLeave[Item] \end{array} \qquad \begin{array}{|l} ABQueueJoin[Item] \\ \Delta RABQueue[Item] \\ RBQueueJoin[Item] \end{array}$$

$$\begin{array}{|l} RABQueueReset[Item] \\ \Delta RABQueue[Item];\ RBQueueReset[Item] \end{array}$$

$$\begin{array}{|l} RABQueueAbandon[Item] \\ \Delta RABQueue[Item];\ item? : Item \\ \hline \exists\, q1, q2 : \operatorname{seq} Item \bullet items = q1 \frown \langle item? \rangle \frown q2 \land items' = q1 \frown q2 \end{array}$$

1.4.2 Global Properties

Class extensions are obtained by instantiating the *SCl* Z generic (see [10]). State extensions of Queue, BQueue, and RBQueue are:

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

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

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

Extension initialisations say that classes have no living instances:

$$SQueueInit[Item] == [SQueue[Item]' \mid stQueue' = \varnothing]$$

$$SBQueueInit[Item] == [SBQueue[Item]' \mid stBQueue' = \varnothing]$$

$$SRBQueueInit[Item] == [SRBQueue[Item]' \mid stRBQueue' = \varnothing]$$

Association HasQueues is represented as a function relating QueueManager objects with sets of Queues indexed by set *QId* (the queue identifier).

$$\begin{array}{|l} AHasQueues \\ rHasQueues : \mathbb{O}\ QueueManagerCl \pfun (QId \pfun \mathbb{O}\ QueueCl) \end{array}$$

The next invariant says that the RBQueue instances held by a QueueManager must have queues of size at most 5.

$$\begin{array}{|l} RBQueuesInHasQueuesSizeLeq5[Item] \\ SystemGblSt[Item] \\ \hline \forall\, oqm : \mathbb{O}\ QueueManagerCl;\ rq : \mathbb{O}\ RBQueueCl \mid oqm \in \operatorname{dom} rHasQueues \bullet \\ \quad rq \in (\operatorname{ran}(rHasQueues\ oqm)) \cap sRBQueue \Rightarrow \#(stRBQueue\ abq).items \leq 5 \end{array}$$

	Relevant outcome	**Issue**
BQueue.join	applicability proof fails	applicability
PBQueue.join	applicability proof fails	applicability
RBQueue.reset	does not refine skip ($\Xi BQueue$)	refinement of skip
ABQueue.abandon	does not refine skip ($\Xi BQueue$)	refinement of skip
JQBQueue.join	correctness proof fails	operation overriding
QueueManager	Global invariant breaches freeness assumption	global Interference

Table 1.1: Results of the BI analysis of the queues example of Fig. 1.8.

1.5 The refinement straight-jacket and some loopholes

The BI proof rules derived in section 1.3 are over-restrictive. Ordinary inheritance hierarchies, such as the queues example of Fig. 1.8, fail to be pure BIs. Furthermore, the rules may be misleading as one may wrongly conclude that inner BI entails overall BI, which is not necessarily the case because global constraints may invalidate what is proved locally (section 1.3.5).

Table 1.1 summarises the BI analysis for the example of figure 1.8. The next sections discuss the four issues that emerged.

1.5.1 Applicability

In Fig. 1.8, class BQueue fails to refine Queue and PBQueue fails to refine BQueue; applicability fails for operation join on both accounts:

- 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.
- Pre-condition of PBQueue.join includes $\#\, items = maxQ - 1 \Rightarrow item? \in privileged$, which does not imply precondition of BQueue.join.

In refinement, the concrete type may weaken the precondition; here, the subclass preconditions are stronger.

These failures happen because the concrete operations strengthen the inherited pre-condition, violating substitutability as the behaviour becomes observably different when the concrete type is used in place of the abstract one. Suppose a braking system of a car; the abstract type says "upon brake slow down" (precondition *true*), and the concrete type says, "upon brake slow down when speed is less than 160 Km per hour" (precondition $speed < 160$) — substitutability cannot possibly hold when pre-conditions are strenthned.

1.5.2 Refinement of skip

Inner BI for `RBQueue` and `ABQueue` also fail with operations `reset` and `abandon` failing to correctness-refine `skip` because both breach `skip`'s ($\Xi BQueue$) constraint saying that inherited state remains unaltered.

1.5.3 Operation overriding

The BI proofs for `JQBQueue.join`, which overrides `BQueue.join`, also fail. The correctness conjecture cannot be proved as `JQBQueue.join` changes the inherited post-condition.

1.5.4 Global interference

The inner BI rules of section 1.3.3 have a local scope (section 1.3.5, fact 6). They can safely be used in contexts where objects of some class hierarchy are not constrained by the environment, following from the *freeness* rule of promotion refinement. However, when *freeness* is breached, BI checks become severely complicated due to the complexity of global proofs.

Global constraint `RBQueuesInHasQueuesSizeLeq5` (section 1.4.2) illustrates this issue. For any client of `QueueManager` that uses its queues, the behaviour of instances of `RBQueue` and its descendants are observably different from the remaining queues. In that context, freeness is breached and the locally proved inner BI no longer holds globally. This global interference is due to divergence of `RBQueue` with respect to the behaviour of other classes in the hierarchy, such as, `BQueue`, `PBQueue` and `JQBQueue`. Suppose that we create, using some `QueueManager`, objects oBQ of class `BQueue` and $oRABQ$ of class `RABQueue` (initially, both queues are empty). If we execute operation `join` five times on them, the observed behaviour is the same. However, a sixth call to `join` on oBQ allows an item to be added to the sequence, but fails on $oRABQ$ because the precondition is breached (the queue already holds five items); substitutability is violated: $oRABQ$ cannot replace of oBQ.

1.6 A BI Examination

Apart from the insufficiently discussed global interference, the problems of the previous section are acknowledged in Liskov and Wing's seminal paper [28]. This section investigates BI in OO programming languages that support design by contract (DbC), namely: JML [15], Eiffel [33] and Spec♯ [27]. DbC

languages are more formal and they support BI following [28], overcoming BI's restrictions through relaxations known as *specification inheritance* [17].

The next sections present (a) BI's support in examined languages, (b) the examination results, and (c) an appraisal of specification inheritance [17].

1.6.1 BI in DbC languages

```
class
   COUNTER
create
   make
feature -- State attributes
   count : NATURAL_8
feature -- constructor
   make
      do
         count := 0
      ensure
         count = 0
      end
feature -- routines
   inc
      require
         count < 127
      do
         count := count +1
      ensure
         count = old count + 1
      end
end
```

(a) Eiffel

```
public class Counter {
   private /*@ spec_public*/ byte count;
   //@ public initially count == 0;
   //@ ensures count == 0;
   public Counter() { count = 0;}
   /*@ requires count < 127;
     @ assignable count;
     @ ensures count == \old(count) + 1;*/
   public void inc() { count++;}
}
```

(b) Java/JML

```
public class Counter {
   [SpecPublic]
    private byte count;
    public Counter()
      ensures count == 0;
   { count = 0;}
    public void inc()
      requires count < 127;
      modifies count;
      ensures count == old(count) + 1;
   { count++; }
}
```

(c) Spec♯

Fig. 1.9: A simple `Counter` class in Eiffel, Java/JML and Spec♯

DbC languages express invariants, contracts made of a pre- and a post-condition and various code checks in the form of assertions. Assertions are used for both static- and run-time verification; the former involves theorem proving and aims to prevent run-time errors. JML is a satellite Java language; assertions are written as program annotations. Eiffel and Spec♯ make assertion specification an integral part of the language.

Figure 1.9 presents a `Counter` class in Eiffel (Fig. 1.9a), Java/JML (Fig. 1.9b) and Spec♯ (Fig 1.9c). The Eiffel assertions of Fig. 1.9a are included in the `require` (pre-condition) and `ensure` (post-condition) clauses. The JML specifications of Fig. 1.9b start with an `@` symbol; the predicates coming after `requires` and `ensures` denote pre- and post-conditions, respectively. Spec♯ uses the same clauses to denote contracts (Fig 1.9c).

In DbC languages, both contracts and invariants are inherited following *specification inheritance* [17]. This is based on two principles of refinement: weakening of precondition and strengthening of postcondition. Inherited operations may be extended or overridden and the accompanying contracts may be combined in the following ways:

- Disjunction of pre-condition of parent and child.
- Conjunction of post-condition of parent and child.

This enables contract extension. JML provides the `also` clause to define new pre- and post-condition pairs; Eiffel provides `require else` and `ensure then` to achieve the same effect. Spec♯ disallows pre-condition extension, however, child invariants may result in extra inherited preconditions; new post-conditions may be added (strengthening) using the usual `requires` clause.

Given a superclass A (abstract), a subclass C (concrete) and an operation Op of A specialised in C, the pre-condition of Op in C is:

$$\textbf{pre}\ A.Op \vee \textbf{pre}\ C.Op$$

The languages differ slightly in the way the subclass post-condition is constructed. Eiffel and Spec♯ use a simple conjunction [33]:

$$\textbf{post}\ A.Op \wedge \textbf{post}\ C.Op$$

JML uses a conjunction of implications [17, 26]:

$$\textbf{pre}\ A.Op \Rightarrow \textbf{post}\ A.Op \wedge \textbf{pre}\ C.Op \Rightarrow \textbf{post}\ C.Op$$

1.6.2 The BI examination proper

The BI examination uses the queues example of Fig. 1.10, a slight variation from Fig. 1.8 specified in Z in section 1.4, with the following changes:

- Generic *Item* is removed as generics are not supported in JML and Spec♯.
- To avoid complications with unbounded queues (in machines things are always bounded), `Queue` is removed and `BQueue` is made the root of the hierarchy.

Table 1.2 presents the tool versions used in the examination carried out in January 2014: version 5.6 of JML compiler (part of the JML tools) and

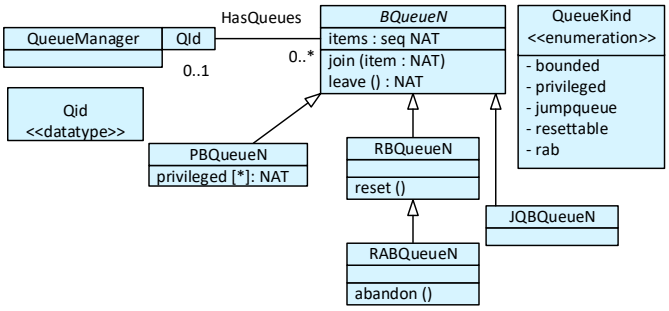


Fig. 1.10: A UML class diagram of an inheritance hierarchy of queues of natural numbers.

	Java/JML	Eiffel	Spec♯
RAC	JML version 5.6_rc4	Eiffel studio 7.3	Spec♯compiler version 1.0.211.26.0 on Microsoft Visual Studio 2010
Proof	ESC/Java 2.0.5, *simplify* theorem prover	N/A	Boogie with theorem prover Z3 version 2.15 (as required by the used version of Spec♯)

Table 1.2: Tool versions used in the BI examination of Java/JML, Eiffel and Spec♯. (Abbreviations: RAC = run-time assertion checking)

version 2.0.5 of ESC/Java[4], which, at the time the examination took place, were more stable than the newer *open JML*. The Eiffel part used release 7.3 of Eiffel studio[5] and Eiffel's static verification[6].

The code implementing the diagram of Fig. 1.10 is given in [4], using an implementation of queues as circular arrays. The examinations consisted of:

- Static assertion verification using theorem proving.
- Runtime assertion checking (RAC) by running code tests.

Table 1.3 summarises the results, indicating, for each test, whether an error was raised or not. The colouring sets this paper's expectation: green (or medium gray) indicates that the result is seen as *correct*, red (or dark gray) denotes *incorrect*, and yellow (or light gray) says that the marked issue

[4] JML tools are available from `http://bit.ly/1aYSdHZ` and ESC/Java 2 from `http://bit.ly/1a4VTUS`.

[5] Available from `http://bit.ly/1iTH8fn`.

[6] The examination used the research version of Eiffel verification environment (EVE), available from `http://bit.ly/1g7avuS`, which includes the auto-proof tool [36] that uses Boogie and the Z3 theorem prover as the verification back-end. Despite many efforts, all attempts to try EVE have failed; EVE's team was contacted but the issue was not solved.

	Java/JML		Eiffel		Spec#	
	proof	runtime	proof	runtime	proof	runtime
PBQueueN.join	no error	no error	N/A	error	no error	error
JQBQueueN.join	error	error	N/A	error	error	error
RBQueueN.reset	no error	no error	N/A	no error	no error	no error
RABQueueN.abandon	no error	no error	N/A	no error	no error	no error
QueueManager	no error	N/A	N/A	error	no error	no error

Table 1.3: Results of the BI examination in JML, Eiffel and Spec♯ using static verification (proof) and runtime assertion checking. Each cell can have the values *no error* (no error raised or signalled), *error raised* and *N/A* (not applicable, because test could not be run). (Green or medium gray = correct. Yellow or light gray = Requiring attention. Red or dark gray = Incorrect.)

deserves further attention. The examination focused on BI's critical points discussed in section 1.5, namely:

- Applicability refinement was examined through class `PBQueueN`, its *privileged items* invariant, and the operation `join`. The results are inconsistent. JML does not signal any errors (statically or at runtime); at run-time, both Eiffel and Spec♯ raise pre-condition violation exceptions; Spec♯'s static checking does not signal any error[7].
- Subclass extra operations altering inherited state is exercised through operations `RBQueueN.reset` and `RABQueueN.abandon`. These are not signalled as errors in any of the examined languages.
- Operation overriding is exercised through operation `JQBQueueN.join`; errors are signalled by all examined languages both statically and at runtime.
- Global interference is exercised here through the class `QueueManager` and its invariant. The static checks do not raise any errors or warnings. Eiffel raises an invariant violation exception at run-time. The runtime test could not be run in JML; Spec♯ does not raise any error.

As mentioned, the DbC languages make certain relaxations to the rules of Liskov and Wing [28]. This is why the results of table 1.3 differ from those of table 1.1, which emerge from the strictest setting. This paper's position concerning the results of table 1.3 is as follows:

- A striking difference from table 1.1 is that the refinements involving the subclass extra operations (`RBQueueN.reset` and `RABQueueN.abandon`) are deemed valid by the DbC languages. This is because the DbC languages use a relaxation that is proved in the next section.
- The results involving `JQBQueueN.join` are consistent with those of table 1.1: it is not a refinement and rightly so.

[7] Spec♯ does not allow preconditions to be added to inherited operations; however, the precondition of the inherited operation `PBQueueN.join` could be specialised through `PBQueueN`'s invariant.

- It is erroneous to deem `PBQueueN.join` as valid, as confirmed statically in JML and Spec♯ and at runtime in JML, as it is not a refinement. This is an issue that stems from the pre-condition rule of specification inheritance [17]. It is this paper's position that an error should be signalled; only the Eiffel and Spec♯ runtime checks are correct. In `PBQueueN` queues, the last place in the queue must be occupied by a privileged item, implying that the subclass precondition is strengthened which causes divergence: when there is only one place left in the queue the superclass allows any item to be added, whereas the subclass (`PBQueueN`) only allows privileged items. The refinement static checks based on proof ignore this problem; it is only Eiffel and Spec♯ that correctly signal this problem at runtime. This highlights an inconsistency: the static checks guarantee absence of such runtime errors, but the tests witness such errors on Eiffel and Spec♯.
- Static verification ignores global interference. Locally-proved BI checks are context-dependent; the user should be informed about contexts that may invalidate the local check. No such warnings were provided by the static checks, but errors were observed at runtime (as invariant violation exceptions) in Eiffel. Spec♯ does not raise an exception. Compilation errors prevented the execution of the test in JML.

The next section looks into specification inheritance [17] to investigate the problem with the pre-condition rule observed in the examination.

1.6.3 A Critique of specification inheritance [17]

The rules of specification inheritance are as follows: (a) pre-condition of parent and child are combined using disjunction (weakening), (b) post-condition of parent and child are combined using conjunction (strengthening). This is contrived; the precondition rule implies that applicability is always true, which may not reflect what the actual precondition says. In the braking example given above — "upon brake slow down" (parent precondition, *true*) and "upon brake slow down when speed is less than 160km/h" (child precondition $speed < 160$) —, the subclass precondition is effectively strengthened, but this is not reflected in the proof which always yields true and should not because the inherited precondition is being strengthened.

1.7 Relaxing the refinement constraints

There are two ways to address the refinement restrictions: (a) we live with the constraints and refactor the OO models to conform to them, or (b) we relax the restrictions. Refactoring seeks to change a model while preserving its meaning and is always a useful remedy. However, in this example, the

sole use of refactoring would mean giving up on inheritance as all classes would have to be merged into a single one. Operations `reset` and `abandon` change abstract state, so they need moving in to the superclass; `BQueue`'s special behaviour requires moving into the superclass; overall, we achieve a valid refactoring at the cost of inheritance's modularity.

Classical refinement, established to cater to stepwise software development, requires that programs or concrete models conform to the more abstract specifications to ensure that vital aspects captured by the abstraction are fulfilled by the more concrete abstraction levels. The relaxations that follow liberate BI by challenging assumptions of refinement. The sequel introduces virtual operations which underpin all three relaxations, followed by an explanation on the particularities of OO abstract classes that are exploited for relaxations before explaining each relaxation.

1.7.1 Virtual operations

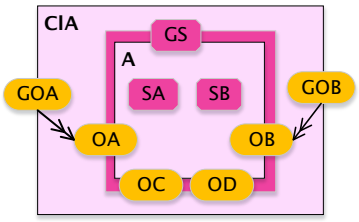


Fig. 1.11: Virtual Operations. Inner operations OC and OD are not promoted and invisible to the class's environment — they are virtual.

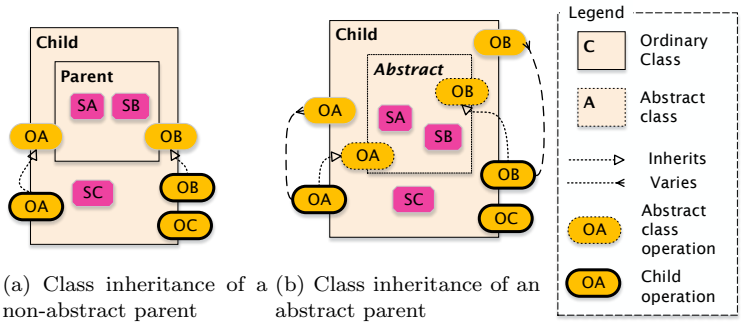
The common remedy proposed here to counter all identified BI malaises (section 1.5) is the notion of *virtual operation*, depicted in Fig. 1.11. An operation is virtual if it is defined in the class's inner type but it is not made available to the environment. In Fig. 1.11, inner operations OC and OD are virtual because they are invisible to the environment as they are not promoted (def. 20).

Virtual operations provide the right conditions to lift the applicability refinement proof obligation. There is no need to comply with the circumstances in which the abstract button may be pressed because it cannot be pressed (it is invisible); correctness, however, remains as the concrete operation needs to comply with the effects of the abstract operation. Formally, any inner operation *co* extension-refines a virtual operation *ao* ($co \sqsupseteq_{Ext} ao$) if and only if refinement-correctness holds — applicability is dropped (def. 21).

1.7.2 The particularities of OO abstract classes

Certain relaxations exploit the particularities of OO *abstract* classes[8], which lack a direct existence as all their instances directly belong to their descen-

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



(a) Class inheritance of a non-abstract parent

(b) Class inheritance of an abstract parent

Fig. 1.12: Inheritance of non-abstract (a) and abstract classes (b).

dant classes (def. 12). Operations of an abstract class are inherently *virtual* as they cannot possibly be executed; they have two purposes: (i) set a template model of behaviour to be varied and specialised by descendants without a commitment to the environment, and (ii) provide polymorphism in the outer view by offering a multitude of possible descendant behaviours chosen dependently on the class of the object on which the operation is called. Figure 1.12 uses the button analogy (rounded-rectangles) to contrast inheritance of non-abstract and abstract classes; the former (Fig. 1.12a) have an actual existence determined by their direct living instances with their buttons being pushable from the environment; abstract parents (Fig. 1.12b), on the other hand, have an indirect existence determined by the living instances of their progeny with their environment-hidden buttons acting as templates to be specialised and elaborated by child classes in the inner view, and those environment-exposed buttons being polymorphic in the outer view.

1.7.3 The child-extra operations relaxation

Subclasses may have extra (or non-inherited) operations. The classical approach to this problem involves a `skip` parent operation doing nothing (section 1.5) and ensuring substitutability: the skip button does nothing with the concrete button doing something but respecting skip. This is too restrictive (section 1.5) as it implies that inherited state cannot be altered.

The relaxation proposed here replaces `skip` with an operation that simulates the concrete operation in the abstract world. This operation is virtual as it is not expected by the environment. For example, when `RBQueue` is used when a `BQueue` is expected, all is needed are operations `join` and `leave`, the simulating substitute of `reset` is invisible to the environment.

Virtual simulating operations are constructed from subclass operations. Given a subclass operation *co* (concrete) and the BI refinement function ϑ

(def. 10), the required simulating abstract operation is (def 22):

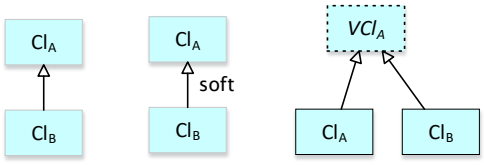
$$ao^{\varnothing} = \vartheta \sim {}_{\S}co\,{}_{\S}\,\vartheta$$

Any concrete operation *co* extension-refines its corresponding abstract virtual operation $ao^{\varnothing}$ — $co \sqsupseteq_{Ext} ao^{\varnothing}$ by fact. 7. Correctness was proved for all cases in the Isabelle proof assistant (further details in [4]) and as $ao^{\varnothing}$ is virtual there is no need to prove applicability. Hence, child extra operations can be added freely: for any concrete operation *co* there is always an abstract virtual operation $ao^{\varnothing}$ that simulates it!

1.7.4 The abstract class relaxation

This relaxation exploits the inner facet of abstract class operations discussed above (section 1.7.2), namely the fact that internally such operations are inherently virtual (def. 20). The relaxation stipulates that the inner BI checks for inherited operations of an abstract class require the correctness proof only as applicability proofs are lifted because the operations are virtual (def. 23).

1.7.5 The soft parent relaxation



(a) Problem (b) Solution (c) Underpinning

Fig. 1.13: The soft parent relaxation.

This relaxation builds up on the abstract class relaxation explained above. It tackles the problem of divergent subclass behaviour, which the abstract class relaxation also tackles, but avoids the need for abstract classes. The soft parent relaxation (Fig 1.13) should be used whenever we need divergent subclass behaviour and a non abstract parent, exemplified in Fig. 1.13a by classes Cl_B and Cl_A respectively; the assertion that the parent is *soft* (Fig. 1.13b) results in an underpinning configuration with a virtual abstract class (VCl_A), an abstraction of the parent class Cl_A, subclassed by both Cl_B and Cl_A (Fig. 1.13c).

The relaxation introduces a virtual abstract superclass behind the scenes, an abstraction of the soft parent, to exploit the divergence offered by abstract classes (Fig. 1.13c). It should be used whenever we need both (i) subclass divergent behaviour, and (ii) parent instantiability. Due to unwanted divergence, it should be used with care. It cannot be applied when the parent is a child of a hard parent (neither abstract nor soft).

1.7.6 The inheritance freeness relaxation

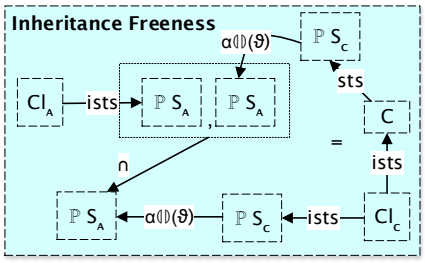


Fig. 1.14: Inheritance freeness.

Global interference occurs whenever global constraints cause divergence between the behaviour of objects in the global space and what would be observed in a locally-confined space. *Inheritance-divergence* concerns the behaviour of child objects becoming observably different from their parent counter-parts. Global interference can be seen as the often inevitable effect of the environment upon the living objects that inhabit it. If inheritance-divergence is caused by global interference, locally proved BIs do not hold globally. The classical condition ensuring global BI-preservation is class freeness (Fig. 1.7, def. 19), which is breached whenever global constraints affect the local spaces of classes. As class freeness is an overkill (fact 6), this paper proposes the *inheritance freeness* relaxation.

Inheritance freeness stipulates that child classes should not be more globally constrained than their *hard* parents (those neither abstract nor soft) to prevent unwanted divergence. This covers two cases: (i) the parent is either abstract or soft and the child is directly affected by global constraints with inheritance divergence being allowed because the parent operation in the outer view is polymorphic; (ii) the parent is hard and all relevant global constraints are expressed in terms of the parent, which implies absence from inheritance divergence because global constraints affect equally both parent and descendants. The relaxation relies on a more relaxed freeness condition, which widens the range of situations in which BIs hold globally as captured by the commuting in Fig. 1.14 (def. 24), resulting in the following equations:

$$\alpha_{(\!|} \; r \; xs = r \; (\!| \; xs \; |\!)$$

$$((\alpha_{(\!|} \; \vartheta) \circ ists) \; Cl_C = ists \; Cl_A \cap ((\alpha_{(\!|} \; \vartheta) \circ sts \circ ists) \; Cl_C$$

Above, function $\alpha_{(\!|}$ applies the relation image to the given relation; it is used to take the relation image of function ϑ (def. 10). The left-hand side of the equation obtains the set of object states of Cl_C and casts them to A using ϑ. The right-hand side obtains the unconstrained objects states and casts them to the abstract inner type A to yield the intersection with the allowed object states of Cl_A. Overall, it says that the set of allowed objects states of Cl_C in Cl_A must be the same as the set of allowed object states in Cl_C.

Hence, applicability-relaxed BI (def. 23) is elaborated to arrive at a relaxed BI definition which replaces class freeness with inheritance freeness (def. 25):

$$vos = vopsids \; Cl_A \lhd ((ops \circ ity) \, Cl_A)$$

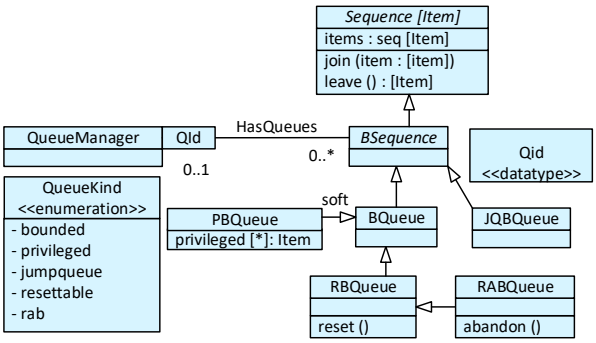


Fig. 1.15: A UML class diagram of an inheritance hierarchy of queues, resulting from a refactoring of the paper's running example given in Fig. 1.8.

$$Cl_C \sqsupseteq_{BI} Cl_A \Leftrightarrow ity\ Cl_C \sqsupseteq_{Ext} ity\ Cl_A \bowtie vos \wedge Cl_A\ \mathbf{Inhfree}\ Cl_C$$

This relaxation implies two things: (a) children may diverge from non-hard parents; (b) if the parent is hard, then inner children may not be directly affected by global constraints. This is translated into a design guideline:

Global Invariants constraining the local definitions of classes that are part of an inheritance hierarchy, should be formulated in terms of classes without hard parents (neither abstract nor soft).

In the queues example of Fig. 1.8 (page 11), any invariants affecting the local definitions of `RBQueue` and `RABQueue` should be stated in terms of the abstract `Queue` or `RBQueue`, the uppermost non-abstract class.

1.7.7 Queues Revisited

The relaxations above would render the queues example of Fig. 1.8 BI-conformant, albeit with a refactoring. Figure 1.15 presents the refactored class model, which is as follows:

- Abstract class `Sequence`, the root of the hierarchy, provides a non-deterministic `join`: an item can be added to either the sequence's front or back. Operation `Sequence.leave` does de-queuing (head of sequence is removed from it). Abstract class `BSequence` puts a bound on the sequence. These classes accommodate `JQBQueue` and its peculiar queue-jumping behaviour which caused correctness-refinement to fail. This paper offers no remedy other than refactoring for correctness-refinement malaises. The refactoring together with the abstract class relaxation (section 1.7.4) would entail: $JQBQueue \sqsupseteq_{BI} BSequence \sqsupseteq_{BI} Sequence$.

- Class `BQueue` restores the normal queue behaviour with elements added to the back of the sequence, and $BQueue \sqsupseteq_{BI} BSequence$ as `BSequence` is abstract. The soft parent relaxation would entail $PBQueue \sqsupseteq_{BI} BQueue$ and retain `BQueue`'s instantiability (`soft` annotation indicates that `BQueue` is a soft parent of `PBQueue`).
- The child-extra operations relaxation (section 1.7.3) implies $RBQueue \sqsupseteq_{BI} BQueue$ and $RABQueue \sqsupseteq_{BI} RBQueue$. `BQueue` provides only two operations, `join` and `leave`; this relaxation allows the addition of `RBQueue.reset` and `ABQueue.abandon` without any proof obligations.
- The inheritance freeness relaxation resolves the global interference issue as the globally interfering constraint is being stated in terms of `RBQueue`, which descends from a non-hard parent.

1.8 Discussion

BI relaxations. With a pair of spectacles focussed on rigour and correctness we see that inheritance induces refinement relations. With another pair focussed on practice and expressibility, we see how refinement's restrictions impair inheritance. The queues running example (Fig. 1.8) shows how trivial inheritance hierarchies fail to be refinements in the strictest sense[9]. Section 1.5 identified the hurdles faced when proving BI. This paper conciliates correctness with both flexibility and expressibility by proposing four relaxations that offer BI without sacrifices to inheritance's flexibility, reuse or capacity for incremental definition.

Virtual operations are the key remedy against applicability issues, which is what lies beneath most refinement hurdles highlighted in section 1.5. Virtual operations belong to the class's inner type but are invisible to the environment. Because they are never executed, the refinement applicability constraint may be lifted — the child operation no longer needs to be applicable whenever the parent operation is also applicable —, implying that child preconditions may be narrowed.

The four relaxations are as follows:

- The child extra operations relaxation allows the addition of extra subclass operations without any proof obligations. For any concrete child extra operation, it is possible to find an abstract simulating operation in the parent, which is virtual and therefore free from applicability restrictions and which always correctness-refine the simulating operation.
- The OO abstract class relaxation caters to subclass behavioural diversity. Because abstract class operations are virtual — an abstract class has no direct instances, implying that its inner operations are invisible to the

[9] The subclassing of an unbounded by a bounded queue is a common specialisation that is not a classical refinement. In general, a bounded type does not refine an unbounded one.

environment —, applicability is lifted and all that is required to prove is correctness. This relaxation is consistent with the view that abstract classes are flexible templates with their objects being *polymorphic*, meaning they are allowed to have a multitude of slightly diverging behaviours. As the queues model and other models in [2] show, with due caution, this relaxation is extremely useful whenever precondition narrowing is required; it enables OO inheritance designs that are flexible, make use of polymorphism and preserve semantic behaviour, enabling a whole range of behavioural diversity whereby the conditions for triggering specialised operations may vary freely. For example, suppose we capture mating as a means of sexual reproduction in mammals; abstract class `Mammal` captures the general notion of mating; the exact circumstances in which this occurs would differ from species to species — the mating circumstances of primates being different from the cetaceans with possibly further diversity at the different sub-species of primate —, however, sub-species must comply with the general mammal notion of mating if there is one.

- The soft parent relaxation caters to both behavioural diversity and reuse. A soft parent entails a virtual abstract class (lacks a direct existence), an abstraction of the soft parent; the parent and all its soft children become children of the virtual abstract class, enabling the application of the abstract classes relaxation. It should be used whenever we need more liberal diversity than the one provided by classical (hard) inheritance and, for the sake of reuse and instantiability, we need to avoid making the parent abstract. When going from the queues example of Fig. 1.8 into the BI-compliant model of Fig 1.15, this relaxation kept `BQueue`'s instantiability and accommodated the peculiar `PBQueue`, which in Fig 1.15 was marked as a soft descendant.
- The inheritance-freeness relaxation tackles global interference. As the queues example highlights, the outside world may constrain the internal states of classes, invalidating locally proved BIs. This relaxation widens the range of situations in which proved BIs hold globally improving upon promotion (or class) freeness [29], which requires the outside world not to constrain the local states of classes. Inheritance-freeness is based on the idea that a child class should not be more globally constrained than its *hard* parents (neither abstract nor soft). It results in a design guideline: global constraints affecting the inner states of classes should not be stated in terms of instantiable classes with hard parents. For example, suppose two sub-species of human and an environment that constrains one of these sub-species to the point that the characteristic human up-right walking is severely limited; in this context, any locally proved BI for walking would only hold in this environment by appeal to this relaxation if class human is made abstract, the sub-species are made soft descendants, or the environment constraint is stated in terms of class human. The paper justifies this relaxation using formal-based argumentation. When this relaxation is

not applicable there is not a practical way to verify BI; global refinement proofs are very complicated even in small systems.

The paper's queues example and other examples from [2] combine refactoring with the paper's relaxations to construct BI hierarchies. All BI conjectures of the queues example were proved in Z/Eves. Usually, proofs at the level of local types are trivial; most of them are automatically provable in Z/Eves.

BI in design by contract languages. This paper examines BI in the design-by-contract (DbC) technologies Java/JML, Eiffel and Spec♯ (section 1.6). It draws attention to the following:

- The precondition rule is a problem in all examined languages (originating from specification inheritance [17]). The examination highlighted inconsistencies between static and runtime verification for a specialised operation that narrows the precondition. The static checks concluded that there were no problems with the pre-condition of operation `PBQUeueN.join`, but the runtime tests triggered pre-condition violations in Eiffel and Spec♯. This is inconsistent; the static checks are there to ensure that such errors do not occur at runtime. A thorough analysis revealed a problem with contract inheritance [17], causing discrepancies between contract and code.
- The examined languages ignore global interference. The paper shows how locally -proved BIs fail to be preserved in all global contexts. DbC languages should warn users of such problems because they may breach substitutability and, hence, cause unexpected run-time errors.

It is interesting to compare the examined languages. Eiffel was the best in runtime verification, but its static verification approach could not be examined as it was being researched at the time of writing and was not part of Eiffel's official release; all efforts to try Eiffel's verification environment failed. Spec♯ was the best language in its support for static verification with good back-end theorem proving based on Boogie and Z3; however, some novel features of Spec♯, tuned for verification, appear to be non-intuitive.

On refinement. The refinement theory founded on total relations [22] is simple and intuitive, but rests on an assumption not applicable to all computing settings: that computing behaviours can be captured by a total relation. Once we cater to partial behaviours the rules become more complicated and they bring out the interesting applicability condition. A large part of the refinement work presented here involves a careful study of applicability to understand the circumstances that allow this rule to be lifted.

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 ADT-based data refinement to classes, where class

refinement equates to Z promotion refinement. Behavioural inheritance is just one specific class refinement because the refinement relation is fixed.

On OO Inheritance. This paper proposes soft inheritance as opposed to the classical hard inheritance to help defining BI hierarchies. Languages concerned with BI could consider the soft annotation to say that particular inheritances are soft (some degree of inheritance-divergence is allowed).

1.9 Related Work

This paper extends the work presented in [3, 2], elaborating and reinforcing the paper's BI relaxations. This extension has four key components:

- The abstract model of OO of section 1.2 results from the insight gained from ZOO, a Z style of OO [2, 10, 3] that builds-up on Hall's work [19, 20]. It represents a class as two ADTs, making it consistent with models of OO programming languages with a formal semantics (like those based on design-by-contract, such as Eiffel, JML and Spec♯); Meyer [32] sees the OO paradigm founded on ADTs with classes having a type view and a module view, which correspond to the inner and outer ADTs that make-up a class in the paper's OO model. This model clearly frames OO within data refinement, contributing to the generalisability of the paper's results, which go beyond Z or ZOO[10].
- The effort on mechanical verification with the Isabelle theorem prover provided insight and feedback, which helped o elaborate the BI relaxations, and made the proof effort more reliable by diminishing the possibility of human error through the use of state of the art proof technology. The derivations of the BI refinement rules have been proved in Isabelle; the child-extra operations relaxation has been proved in Isabelle also.
- The novel soft parent relaxation and the accompanying notion of soft inheritance.
- The paper's BI examination of the three design by contract (DbC) languages, JML, Eiffel and Spec♯, which is entirely new. It highlights a problem in the precondition rule that stems from [17].

The paper addressees the tension between the constraints of formal refinement and the practical needs of software engineering [13]. Retrenchment [13, 14] is a more liberal formal-refinement approach that tackles this problem. This liberalisation idea drives the paper's relaxations. However, the paper's approach to relaxation differs from retrenchment; whilst retrenchment provides conjectures that give room for narrowing the pre-condition and widening the post-condition, this paper follows the refinement tradition

[10] The paper's model, differs from first-order models, such as Alloy's [25], which represents class attributes as relations with the resulting models being global and flat.

of deriving rules from a more general setting, which are then further analysed with respect to substitutability to produce relaxations.

Dhara da Leavens [17] used subclass extra operations BI relaxation. However, this relaxation is neither proved nor justified on formal grounds in [17]. To the author's knowledge, this paper provides the first mechanical proof of this relaxation carried out using proof technology to reinforce its foundations.

Abrial [1] proposes keep operations to overcome the restrictions of the skip approach. keep operations are non-deterministic and guaranteed to preserve the invariant; they may be safely added to abstract types [1]. They resemble simulating abstract operations used in the child-extra operations relaxation, which are safe because they are not visible to the environment.

While the OO model of Liskov and Wing [28] is similar to ZOO's (there is a mapping from objects to their state), their approach is based on a earlier method of data refinement [23] that does not consider initialisation and finalisation. This paper uses data refinement based on simulation [22], the enduring basis of the theory, which accounts for object creation (initialisation) and deletion (finalisation); BI cannot be guaranteed if these are not checked. The rules of [28] correspond to the rule for blocking refinement given in [4]. In [28], the BI overkill is highlighted, but relaxations are not considered.

Wehrheim and Fischer [18, 37] investigate BI 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 [34, 16] defines a formal semantics for inheritance and a notion of class refinement, but a discussion of BI is generally absent in its books. In [16], BI and its relation to refinement is discussed, but no proof obligations are proposed to check its correctness.

1.10 Conclusions

This paper investigates behavioural inheritance (BI) [28] by building up on insight gained from previous work [2, 3]. It delves into BI's foundations through an abstract mathematical model of object orientation to come up with sound ways of reconciling the correctness-sensitive refinement facet of BI with the flexibility that characterises inheritance and the object-oriented

(OO) paradigm. The paper's accompanying technical report [4] provides further details not given here; the Isabelle proofs supporting the work presented here are given in [5]; all BI proofs related to the Z models corresponding to Figs. 1.8 and 1.15 were undertaken in the Z/Eves Z prover.

BI's constrictions are known since BI's inception [28]. As this paper shows, existing BI relaxations [17] are problematic and unconvincing; some relaxations are over-permissive (everything is a BI) and untrue to refinement, resulting from an over-simplification of an intricate reality. This paper revisits BI by clearly framing it within the theory of data refinement [22]; it derives BI proof rules from the foundations of the theory whilst trying to relax the refinement constraints that severely limit BI with respect to an ethos of inheritance and OO characterised by flexibility.

The mathematical model of OO developed in section 1.2 supports the framing of BI under the umbrella of data-refinement [22] whilst providing a mathematical foundation to the paper's major contributions. Data refinement is simply and elegantly expressed as sub-setting in the world of *total* relations. If totality is appropriate for the concrete computing world of programs where machine processing requires everything to be defined, it becomes inadequate for the higher-levels of abstraction focussed on the essence of problems and which, for the sake of abstraction, demand *partial* relations. Furthermore, design by contract (DbC) languages support a computing paradigm based on explicitly declared assumptions (pre-conditions) and expectations (post-conditions) which requires partiality. To accommodate partiality and BI, the paper adapts the rules of data refinement to partial relations specific to BI. To accommodate object orientation, the paper takes the theory of data refinement for abstract data types (ADTs) and adapts it to OO classes. The abstract OO model of section 1.2 based on ADTs enables the derivation of BI-specific conjectures from the theory of data of refinement.

The BI conjectures of section 1.3 are derived through the totalisation technique used in Z refinement. The simple refinement relation (ϑ function, def. 10) capturing inheritance's child-parent relation reflects the fact that there are actually two ADTs involved in a class construction known in Z as promotion, and this provides a separation: the inner and outer types as representative of the corresponding views or worlds of a class. Most derivation work was grounded on the ϑ refinement function, which is confined to the inner type. By exploiting the properties of ϑ, the paper proved that the inner BI conjectures reduce to a single set (corollary 1). From here, the paper derived the actual inner BI conjectures (or ADT extension refinement) in the relational (fact 4) and Z-schema settings (fact 5), alerting that overall BI refinement requires class freeness (fact 6).

The derived BI conjectures of section 1.3 were applied to the paper's running example in section 1.5, which endorsed what is long known: trivial inheritance relations widely used in OO programming are not refinements in the strictest sense. The analysis of section 1.5 identified issues with child-extra operations, operation overriding, child operations that strengthen the

pre-condition and global interference. Global interference, neglected in the literature, is the realisation that locally proved BIs do not necessarily hold globally. The analysis resulting from the paper's running example poses two questions: (i) how to reconcile BI with an ethos of inheritance characterised by flexibility? (ii) How do DbC languages deal with BI's over-restrictiveness?

The examination of the DbC languages Eiffel, JML and and Spec♯ of section 1.6 highlighted inconsistencies between static and dynamic verification, a neglect for global interference, and issues with the pre-condition conjecture of [17], which is over-permissive (everything is a refinement) validating statically BIs that turned out to be invalid at run-time.

The BI analysis (sections 1.5 and 1.6) motivates the quest for improved relaxations. The paper's relaxations target two issues identified in section 1.5, namely: applicability's restrictions and global interference. The core remedy against applicability malaises is *virtual operations*, which result from the observation that certain inner class operations are invisible to the environment (hence, virtual), giving room to lift the applicability rule: the child operation no longer needs to be applicable whenever the parent operation is applicable because the parent operation is invisible to the environment. This idea underpins all the paper's relaxations. For global interference, this paper take as normative the restriction of promotion refinement's freeness constraint, which is relaxed through the novel inheritance freeness based on the idea that children should not be more constrained than their hard parents. Together with refactoring, the relaxations were used to transform the queues running example of Fig. 1.8 into the BI compliant alternative of Fig. 1.15.

This paper has the following contributions:

- The useful abstract class relaxation and the underlying notion of virtual operations. Although, they were both proposed in [2, 3], they have been elaborated here. This paper emphasises virtual operations and the fact that applicability is lifted whenever a virtual operation is refined.
- The novel soft parent relaxation and the accompanying notion of soft inheritance, which is useful in situations requiring both inheritance-divergence and parent instantiability.
- The reinforcement of the child-extra operations relaxation, which is also proposed in [2, 3] and [17], by carrying out the required formal proofs in the state of the art Isabelle prover; this increases the reliability of proof by diminishing the possibility of human error. Using the virtual operation notion, the applicability proof obligation is lifted; it was proved that child extra operations always correctness-refine their corresponding simulating abstract operations — hence, child extra operations may be added freely.
- The inheritance-freeness relaxation dealing with global interference elaborates on [2, 3]. Due to its complexity, this relaxation was proved informally using formal argumentation in the accompanying technical report [4].
- The critique to the specification inheritance relaxations [17] grounded on the examination of three DbC languages relying on [17]. The pre-condition

(or applicability) conjecture of [17] as being problematic with the examination highlighting inconsistencies between static and dynamic verification.

Acknowledgements

The work presented here commenced when I was a PhD student at the University of York supervised by Prof. Susan Stepney and Dr. Fiona Polack. I am very happy that this paper is part of this volume because Susan Stepney was immensely influential and inspiring not only to the whole work presented here, but to myself and my academic work.

Appendix 1.A: Mathematical Definitions

Definition 1 (Given Sets)**.** Given sets (all disjoint) O, S, I and E represent objects, states, identifiers and environments respectively. □

Definition 2 (Sets for ADTs)**.** The following definitions support abstract data types (ADTs):

- Set $Init = E \leftrightarrow S$, of relations between environments E (def. 1) and initial states S (def. 1), represents all possible ADT initialisations (initial states).
- Set $Fin = S \leftrightarrow E$, of relations between final states S (def. 1) and environments E (def. 1), represents all possible ADT finalisations (final states).
- Set $Op = I \nrightarrow (S \leftrightarrow S)$, of functions from identifiers (def. 1) of operations (names), to a relation between states (representing a transition between a a before- and an after-state), represents all possible ADT operations.

□

Definition 3 (ADTs)**.** An abstract data type (ADT) $t = (s, i, f, os)$ is made up of set $s \subseteq S$ of all type states (def. 1), an initialisation $i : Init$ (def. 2), a finalisation $f : Fin$ (def. 2), and an indexed set of operations $os : Op$ (def. 2).

The set of ADTs, such that $t : ADT$, is defined as:

$$ADT = \{(s, i, f, os) \mid s \in \mathbb{P}\, S \land i \in E \leftrightarrow s \land f \in s \leftrightarrow E \land os \in I \nrightarrow (s \leftrightarrow s)\}$$

Auxiliary Definitions. Several functions extract information from an ADT:

$$sts\,(s, i, f, os) = s \;\; init\,(s, i, f, os) = i \;\; fin\,(s, i, f, os) = f \;\; ops\,(s, i, f, os) = os$$

□

Definition 4 (Sets for Classes)**.** The following sets support classes:

- Set $OSt = O \nrightarrow S$, of partial functions from an object to a state, represents the state of a class: its living objects are mapped to their states.

- Set $ONew = E \leftrightarrow (OST \leftrightarrow OST)$, of relations between environments E and a transition of class global states $(OST \leftrightarrow OST)$, represents the effect of a class constructor on the global class state .
- Set $ODel = (OST \leftrightarrow OST) \leftrightarrow E$, of relations between a transition of class global states $(OST \leftrightarrow OST)$ and environments E, represents the effect of a class destructor on the global class state.
- Set $OUpd = I \nrightarrow I \nrightarrow (OST \leftrightarrow OST)$, of partial functions from identifiers of global operations to another function from identifiers of local operations to a transition of class global states (described as a relation, $OST \leftrightarrow OST$), which represents the effect of a class update operation: the operation results in a transition from a particular class state to another class state in which the state of the affected object is updated through a local operation.

□

Definition 5 (Promoted Operations)**.** Partial functions $nOps : ADT \times OST \nrightarrow ONew$, $dOps : ADT \times OST \nrightarrow ODel$, and $uOps : ADT \times OST \nrightarrow OUpd$ give constructor, destructor and modifier operations of a class, respectively, which are built from the given inner type (see defs. 3 and 4 for the used sets). The functions are as follows:

$$nOps = (\lambda\, t : ADT;\ ost : OST \bullet \{nop : ONew \mid \exists\, e : E;\ o : O \setminus \mathrm{dom}\, ost;\ s : S \mid (e, s) \in init\ t \bullet e \mapsto \{(ost, ost \cup \{o \mapsto s\})\} \in nop\})$$

$$dOps = (\lambda\, t : ADT;\ ost : OST \bullet \{dop : ODel \mid \exists\, e : E;\ o : \mathrm{dom}\, ost \mid (ost\ o, e) \in fin\ t \bullet \{(ost, \{o\} \lhd ost)\} \mapsto e \in dop\})$$

$$uOps = (\lambda\, t : ADT;\ ost : OST \bullet \{uop : OUpd \mid \exists\, i_g, i_l : I;\ o : \mathrm{dom}\, ost;\ s' : sts\ t \mid i_l \mapsto \{(ost\ o, s')\} \in ops\ t \bullet i_g \mapsto \{i_l \mapsto \{(ost, ost \oplus \{o \mapsto s'\})\}\} \in uop\})$$

□

Definition 6 (Class)**.** A class is $C = (ci, t, is, os, stm, ocl, c, d, ms)$ is made-up of a class identifier $ci : I$, type $t : ADT$ (def. 3), a set of possible object states $is \subseteq sts\ t$, a set of objects $os \subseteq O$ (all possible class objects), a global class state $stm \in OST$, a mapping $ot : os \nrightarrow I$ relating objects to the identifiers of their direct classes, a constructor $c \in ONew$, a destructor $d \in ODel$ and modifiers $ms \in OUpd$ (see def. 4 for $ONew$, $ODel$ and $OUpd$).

The set of classes, such that $C : Cl$ is defined as:

$$Cl = \{(ci, t, is, os, stm, ot, c, d, ms) \mid ci \in I \land t \in ADT \land is \subseteq sts\ t \land os \in \mathbb{P}\, O \land stm \in os \nrightarrow is \land ot \in os \nrightarrow I \land \mathrm{dom}\, ot = \mathrm{dom}\, stm \land c \in nOps\,(t, stm) \land ms \in uOps\,(t, stm) \land d \in dOps\,(t, stm)\}$$

Above, it is asserted that both the global class state function stm and the typing mapping ot are defined for the set of living objects of the class.

Auxiliary Definitions. Several functions extract information from a class:

$$icl\,(ci, t, is, os, stm, ot, c, d, ms) = ci \qquad ity\,(ci, t, is, os, stm, ot, c, d, ms) = t$$
$$ists\,(ci, t, is, os, stm, ot, c, d, ms) = is \qquad osu\,(ci, t, is, os, stm, ot, c, d, ms) = os$$
$$los\,(ci, t, is, os, stm, ot, c, d, ms) = \mathrm{dom}\ stm \qquad ost(ci, t, is, os, stm, ot, c, d, ms) = stm$$
$$ocl(ci, t, is, os, stm, ot, c, d, ms) = ot \qquad cop\,(ci, t, is, os, stm, ocl, c, d, ms) = c$$
$$dop\,(ci, t, is, os, stm, ocl, c, d, ms) = d \qquad mops\,(ci, t, is, os, stm, ocl, c, d, ms) = ms$$
$$pops\,(ci, t, is, os, stm, ocl, c, d, ms) = \bigcup\{im : \mathrm{dom}\ ms \bullet \mathrm{dom}(ms\ im)\}$$

Above, *icl* gives the class's identifier; *ity* yields the inner type, *ists* yields the set of object states; *osu* yields the universe of objects of the class (all possible class objects); *los* yields the living (or existing) objects of the class; *ost* yields the class's global state; *ocl* gives the mapping from living objects into the identifiers of their direct classes; *cop*, *dop* and *mops* yield the constructor, destructor, and modifier operations, respectively; and *pops* yields the set of promoted inner type operations. □

Definition 7 (State Space Composition)**.** Function $\omega : \mathbb{P}\,S \times \mathbb{P}\,S \to \mathbb{P}\,S$ builds larger state spaces from smaller ones. Given states spaces $S_1, S_2 : \mathbb{P}\,S$, $\omega(S_1, S_2)$ yields their composition. □

Definition 8 (State Extension)**.** Given $C, A : ADT$ (def. 3), C is a state extension of A if and only if C's state space is made up of A's with something extra (see def. 7 for ω):

$$C\ \mathbf{extends}\ A \Leftrightarrow \exists\, S_x \in \mathbb{P}\,S \mid sts\ C = \omega(sts\ A, S_x)$$

□

Definition 9 (Z schema state extension)**.** Given ADTs $C, A : ADT$ (def. 3), defined as Z schemas, C is a state extension of A as per def. 8 if it is defined using the following Z schema calculus formula:

$$C == A \wedge X$$

This says, using the Z schema conjunction operator (equivalent to ω of def. 7), that the state of C is that of A with something extra (X). □

Definition 10 (ϑ-function)**.** Given $C, A : ADT$ (def. 3), such that C **extends** A (def. 8), function $\vartheta : sts\ C \to sts\ A$ yields the state being extended:

$$\vartheta\,(\omega(sts\ A, S_x)) = sts\ A$$

This recovers the state space that gave rise to the ADT's compound state.

Function $t\vartheta : ADT \times ADT \to (\mathbb{P}\,S \to \mathbb{P}\,S)$, described as a definite description (operator μ), yields a ϑ given two ADTs:

$$t\vartheta(C, A) = \mu\,\vartheta : sts\ C \to sts\ A \mid C\ \mathbf{extends}\ A \wedge \vartheta\,(\omega(sts\ A, S_x)) = sts\ A$$

□

Definition 11 (Inheritance)**.** Given $Cl_C, Cl_A : Cl$ (def. 6), Cl_C (child) inherits from Cl_A (parent) if and only if the inner state space of Cl_C extends

that of Cl_A (def. 8), the set of objects of Cl_C is a subset of that of Cl_A, and there exists a function ϑ (def. 10):

$$Cl_C \textbf{ inh } Cl_A \Leftrightarrow ity\ Cl_C \textbf{ extends } ity\ Cl_A \wedge osu\ Cl_C \subseteq osu\ Cl_A \wedge los\ Cl_C \subseteq los\ Cl_A \\ \wedge \exists \vartheta \mid \vartheta = t\vartheta(ity\ Cl_C, ity\ Cl_A) \wedge (ost\ Cl_p) \circ \mathrm{id} = \vartheta \circ (ost\ Cl_c)$$

Above, the equation with function composition (symbol $\circ$) captures the diagram commutativity of Fig. 1.3b. □

Definition 12 (Abstract Class)**.** An abstract class lacks direct instances. Set of abstract classes $ACl \subseteq Cl$, a subset of Cl (def. 6), is defined as:

$$ACl = \{ c : Cl \mid (ocl\ c) \sim (\!|\{icl\ c\}|\!) = \varnothing \}$$

This says that the living instances preclude direct instances. □

Definition 13 (Programs)**.** A program is a sequence of operations upon a data type. A complete program begins with an initialisation and ends with a finalisation. A program with operations i_1, i_2 over a type $T : ADT$ (def. 3), results in the complete program: $init\ T \fatsemi ops\ T\ i_1 \fatsemi ops\ T\ i_2 \fatsemi fin\ T$.

Function $cmpops : ADT \times \mathrm{seq}\ I \pfun (S \rel S)$ yields the overall state transition (a relation between states):

$$cmpops(T, \langle\rangle) = id$$

$$cmpops(T, \langle i \rangle \frown is) = (ops\ T\ i) \fatsemi cmpops\ (T, is)$$

Above $\fatsemi$ is relation composition.

Function $cprog : ADT \times \mathrm{seq}\ I \pfun E \rel E$, computes the effect of a complete program, yielding a relation between environments corresponding to the environment expectations and the environment effects of the program:

$$cprog(T, is) = (init\ T) \fatsemi cmpops(T, is) \fatsemi (fin\ T)$$

□

Definition 14 (Data Refinement)**.** For $A, C : ADT$ (def 3), C refines A ($C \sqsupseteq A$) if and only if for each finite sequence of operations os over the indexing set I common to both C and A ($\mathrm{dom}(ops\ A) = I = \mathrm{dom}(ops\ C)$), we have that (*cprog* is as per def 13):

$$cprog\ (C, os) \subseteq cprog\ (A, os)$$

□

Definition 15 (Forwards Simulation)**.** Given $A, C : ADT$ (def 3) and a relation relating their state spaces $r : sts\ A \rel sts\ C$, we say that C refines A ($C \sqsupseteq A$) with r being a forwards simulation if:

$$(init\ C) \subseteq (init\ A) \fatsemi r$$
$$r \fatsemi (fin\ C) \subseteq (fin\ A)$$
$$r \fatsemi (ops\ C\ io) \subseteq (ops\ A\ io) \fatsemi r$$

Above, io is an operation of A and C — $io \in \mathrm{dom}(ops\ A) \cap \mathrm{dom}(ops\ C)$. The rules are as per the commuting of Fig. 1.4 when the relation is upwards. □

Definition 16 (Backwards Simulation)**.** Given $A, C : ADT$ (def. 3) and a relation relating their state spaces $s : sts\ C \leftrightarrow sts\ A$, we say that C refines A ($C \sqsupseteq A$) with s being a backwards simulation if:

$$(init\ C) \fatsemi s \subseteq (init\ A)$$
$$(fin\ C) \subseteq s \fatsemi (fin\ A)$$
$$(ops\ C\ io) \fatsemi s \subseteq s \fatsemi (ops\ A\ io)$$

Above, io is an operation of A and C — $io \in \mathrm{dom}(ops\ A) \cap \mathrm{dom}(ops\ C)$. The rules are as per the commuting of Fig. 1.4 when the relation is downwards. □

Fact 1 (Forwards Simulation, partial relations)**.** Given $A, C : ADT$ (def 3) and a relation relating their state spaces $r : sts\ A \leftrightarrow sts\ C$. In a setting of non-communicating partial operations, C refines A ($C \sqsupseteq A$) and r is a forwards simulation in the non-blocking (or contractual) setting if:

$$(init\ C) \subseteq (init\ A) \fatsemi r \qquad (initialisation)$$
$$r \fatsemi (fin\ C) \subseteq (fin\ A) \qquad (finalisation)$$
$$r \ltimes \mathrm{dom}(ops\ A\ io) \rtimes \subseteq \mathrm{dom}(ops\ C\ io) \qquad (applicability)$$
$$\mathrm{dom}(ops\ A\ io) \lhd r \fatsemi (ops\ C\ io) \subseteq (ops\ A\ io) \fatsemi r \qquad (correctness)$$

Above, io is an operation of A and C — $io \in \mathrm{dom}(ops\ A) \cap \mathrm{dom}(ops\ C)$.

Proof. These rules are proved in [39] and [16] from the rules for total relations (def. 15) using a totalisation technique (see [39] for details). □

Fact 2 (Backwards Simulation, partial relations)**.** Given $A, C : ADT$ (def 3) and a relation relating their state spaces $s : sts\ C \leftrightarrow sts\ A$. In a setting of non-communicating partial operations, C refines A ($C \sqsupseteq A$) and s is a backwards simulation in the non-blocking (or contractual) setting if:

$$(init\ C) \fatsemi s \subseteq (init\ A) \qquad (initialisation)$$
$$(fin\ C) \subseteq s \fatsemi (fin\ A) \qquad (finalisation)$$
$$\mathrm{dom}(ops\ C\ io) \subseteq \mathrm{dom}(s \rhd (\mathrm{dom}(ops\ A\ io))) \qquad (applicability)$$
$$\mathrm{dom}(s \rhd (\mathrm{dom}(ops\ A\ io))) \lhd (ops\ C\ io) \fatsemi s \subseteq s \fatsemi (ops\ A\ io) \qquad (correctness)$$

Above, io is an operation of both A and C — $io \in \mathrm{dom}(ops\ A) \cap \mathrm{dom}(ops\ C)$.

Proof. These rules are proved in [39] and [16] from the rules for total relations (def. 16) using a totalisation technique (see [39] for details). □

Definition 17 (Extension refinement function, general)**.** Let $C, A : ADT$ (def. 3) such that C **extends** A (def 8), the refinement function ϑ is obtained from function $t\vartheta$ (def. 10):

$$\vartheta = t\vartheta(C, A)$$

□

Definition 18 (Extension refinement function, schemas)**.** Let $A, C : ADT$ be two inner Z schema ADTs such that C extends A as described by the Z schema calculus formula $C == A \wedge X$. The ϑ BI function of def. 10 is described by the Z formula:

$$\vartheta = \lambda\, C \bullet \theta\, A$$

This casts the general ϑ of def. 17 into the context of the Z schema calculus. □

Fact 3 (Single set of refinement rules)**.** Given $C, A : ADT$ (def 3), if the refinement relation is a total function $f : sts\ C \rightarrow sts\ A$, then data refinement reduces to a single set of rules with forwards simulation (def 15) being equivalent to backwards simulation (def 16):

$$(init\ C) \subseteq (init\ A) \fatsemi f^{\sim} \Leftrightarrow (init\ C) \fatsemi f \subseteq (init\ A)$$
$$f^{\sim} \fatsemi (fin\ C) \subseteq (fin\ A) \Leftrightarrow (fin\ C) \subseteq f \fatsemi (fin\ A)$$
$$f^{\sim} \fatsemi (ops\ C\ io) \subseteq (ops\ A\ io) \fatsemi f^{\sim} \Leftrightarrow (ops\ C\ io) \fatsemi f \subseteq f \fatsemi (ops\ A\ io)$$

Above, io is an operation identifier defined in concrete and abstract type — $io \in \mathrm{dom}(ops\ A) \cap \mathrm{dom}(ops\ C)$.

Proof. The equivalences were proved in Isabelle. Hand-written proofs are given in [4]. □

Corollary 1 (Single Set of Rules for Extension Refinement)**.** The rules of extension refinement reduce to a single set. This results from applying fact 3 to the BI refinement functions of defs. 17 (relations) and 18 (schemas). □

Fact 4 (Extension Refinement, relations)**.** Given $C, A : ADT$ (def 3), such that C **extends** A (def 8), C extension-refines A ($C \sqsupseteq_{Ext} A$) with ϑ being the refinement function (def. 17) if the three major data refinement conditions — initialisation (init), finalisation (fin) and operations (ops) — are satisfied:

$$\vartheta = t\vartheta(C, A) \qquad C \sqsupseteq_{Ext} A \Leftrightarrow C \sqsupseteq^{init}_{Ext} A \wedge C \sqsupseteq^{fin}_{Ext} A \wedge C \sqsupseteq^{ops}_{Ext} A$$

We define the initialisation and refinement conditions as:

$$C \sqsupseteq^{init}_{Ext} A \Leftrightarrow (init\ C) \subseteq init\ A \fatsemi \vartheta^{\sim} \quad (initialisation)$$
$$C \sqsupseteq^{fin}_{Ext} A \Leftrightarrow \vartheta^{\sim} \fatsemi (fin\ C) \subseteq (fin\ A) \quad (finalisation)$$

An operation extension-refines another if the applicability (appl) and correctness (corr) conditions are met:

$$C \sqsupseteq^{ops}_{Ext} A \Leftrightarrow \forall\, io : \mathrm{dom}(ops\ C) \bullet (ops\ C\ io) \sqsupseteq^{appl}_{Ext} (ops\ A\ io) \wedge (ops\ C\ io) \sqsupseteq^{corr}_{Ext} (ops\ A\ io)$$
$$co \sqsupseteq^{appl}_{Ext} ao \Leftrightarrow \mathrm{dom}(\vartheta \fatsemi ao) \subseteq \mathrm{dom}(co) \qquad (applicability)$$
$$co \sqsupseteq^{corr}_{Ext} ao \Leftrightarrow \mathrm{dom}\ ao \lhd (\vartheta^{\sim} \fatsemi co \fatsemi \vartheta) \subseteq ao \ (correctness)$$

Proof. The rules above were proved in Isabelle (see [4] for further details). □

Fact 5 (Extension Refinement, schemas)**.** Given Z schema inner ADTs $C, A : ADT$ (def 3) such that C **extends** A (def 8) — which in Z is described as $C == A \wedge X$ (where X is the extension) —, then $C \sqsupseteq_{Ext} A$ if and only if:

$$\begin{array}{ll} \forall\, C' \bullet CI \Rightarrow AI & (initialisation) \\ \forall\, C \bullet CF \Rightarrow AF & (finalisation) \\ \forall\, C;\ i? : V \bullet \mathbf{pre}\ AO \Rightarrow \mathbf{pre}\ CO & (applicability) \\ \forall\, C';\ C;\ i?, o! : V \bullet \mathbf{pre}\ AO \wedge CO \Rightarrow AO & (correctness) \end{array}$$

If the finalisation is total (the ADT does not have a finalisation condition) the finalisation rule reduces to *true*.

Proof. The rules above were proved in Isabelle (see [4] for further details). □

Definition 19 (Free Classes)**.** Given any class $Cl_B : Cl$ (def. 6), we say that Cl_B is free (or that the underlying promotion is free) if the following holds:

$$\mathbf{free}\ Cl_B \Leftrightarrow ists\ Cl_B = (sts \circ ity)\ Cl_B$$

This says that the inner states are free from global constraints (commuting of Fig. 1.7). □

Fact 6 (BI Refinement)**.** Let $Cl_A, Cl_C : Cl$ (def 6) such that Cl_C **inh** Cl_A (def. 11). Cl_C is BI conformant with Cl_A ($Cl_C \sqsupseteq_{BI} Cl_A$) if and only if:

$$Cl_C \sqsupseteq_{BI} Cl_A \Leftrightarrow ity\ Cl_C \sqsupseteq_{Ext} ity\ Cl_A \wedge free\ Cl_C \wedge free\ Cl_A$$

This requires that Cl_C's inner type extension-refines Cl_A's and both Cl_C and Cl_A are free (def. 19).

Proof. This applies promotion refinement and freeness of [29] to BI. □

Definition 20 (Virtual operations)**.** Function $vopids : Cl \nrightarrow \mathbb{P}\, I$ identifies the set of possible inner virtual operations of a class (set Cl, def 6):

$$vopids\ cl = \begin{cases} (\mathrm{dom} \circ ops \circ ity)\ cl \setminus pops\ cl & \mathbf{if} cl \notin A Cl \\ (\mathrm{dom} \circ ops \circ ity)\ cl & \mathbf{otherwise} \end{cases}$$

Above, *vopids* yields all identifiers of inner type operations not being promoted or the identifiers of all inner operations if the class is abstract.
□

Definition 21 (Extension Refinement with virtual operations)**.** Extension refinement (def. 4) is extended to cater to virtual operations $vops \subseteq I$. Given $C, A : ADT$ (def 3), such that C **extends** A (def 8), C extension-refines A with virtual operations $vops$ ($C \sqsupseteq_{Ext} A \bowtie vops$) if the following holds:

$$C \sqsupseteq_{Ext} A \bowtie vops \Leftrightarrow vops \subseteq (\mathrm{dom}\ ops\ A) \wedge C \sqsupseteq^{init}_{Ext} A \wedge C \sqsupseteq^{fin}_{Ext} A \wedge (C \sqsupseteq^{ops}_{Ext} A \bowtie vops)$$

The conditions for the refinement of operations then becomes:

$$C \sqsupseteq^{ops}_{Ext} \bowtie vops \Leftrightarrow$$
$$\forall\, io : \mathrm{dom}(ops\ C) \setminus vops \bullet (ops\ C\ io) \sqsupseteq^{appl}_{Ext} (ops\ A\ io) \wedge (ops\ C\ io) \sqsupseteq^{corr}_{Ext} (ops\ A\ io)$$
$$\wedge\ \forall\, io : \mathrm{dom}(ops\ C) \cap vops \bullet (ops\ C\ io) \sqsupseteq^{corr}_{Ext} (ops\ A\ io)$$

This says that correctness suffices to prove that a concrete operation refines a virtual operation. All other definitions are as per fact 4. □

Definition 22 (Simulating abstract operations)**.** Given $C, A : ADT$ (def. 3), such that C **extends** A (def. 8), then for any child (or concrete) operation $co \in ops\ C$, it is possible to calculate a simulating operation $ao^{\varnothing}$, which simulates co in the abstract world, using function $t\vartheta$ (def. 10):

$$\vartheta = t\vartheta(C, A) \qquad ao^{\varnothing} = \vartheta^{\sim} \fatsemi co \fatsemi \vartheta$$

Operation $ao^{\varnothing}$ is virtual.
□

Fact 7 (Refinement of virtual operations)**.** Given $C, A : ADT$ (def. 3), such that C **extends** A (def. 8), then any concrete operation $co \in ops\ C$ extension-refines the simulating operation $ao^{\varnothing}$ (def. 22) — $co \sqsupseteq_{Ext} ao^{\varnothing}$.

Proof. Correctness was proved in Isabelle (further details in [4]). Applicability is dismissed by the virtual operation principle captured in def. 21. □

Corollary 2 (BI of child extra operations)**.** Let $Cl_A, Cl_C : Cl$ (def 6) such that Cl_C **inh** Cl_A (def. 11). Any child extra operation co not defined in Cl_A is inner BI conformant as a result of fact 7. □

Definition 23 (Applicability-relaxed BI)**.** Given $Cl_A, Cl_C : Cl$ (def 6) such that Cl_C **inh** Cl_A (def. 11). Cl_C is BI conformant with Cl_A ($Cl_C \sqsupseteq_{BI} Cl_A$) when the following holds:

$$vos = vopsids\ Cl_A \lhd ((ops \circ ity)\, Cl_A)$$
$$Cl_C \sqsupseteq_{BI} Cl_A \Leftrightarrow (ity\ Cl_C \sqsupseteq_{Ext} ity\ Cl_A \bowtie vos) \wedge free\ Cl_C \wedge free\ Cl_A$$

Above, vos holds virtual operations of Cl_A. □

Definition 24 (Inheritance freeness)**.** Let $Cl_C, Cl_A : Cl$ (def. 6) such that Cl_C **inh** Cl_A (def. 11), ϑ be the function relating inner states of Cl_C and Cl_A (def. 10) and $\alpha_{(\!|\,|\!)}$ be a function that applies the relation image ($(\!|\,|\!)$):

$$\vartheta = t\vartheta(ity\ Cl_C, ity\ Cl_A) \qquad \alpha_{(\!|\,|\!)}\ r\ xs = r\ (\!|\ xs\ |\!)$$

We say that this inheritance relation is free if the following holds:

$$Cl_C\ \mathbf{Inhfree}\ Cl_A \Leftrightarrow ((\alpha_{(\!|\,|\!)}\ \vartheta) \circ ists)\ Cl_C = ists\ Cl_A \cap ((\alpha_{(\!|\,|\!)}\ \vartheta) \circ sts \circ ists)\ Cl_C$$

This says that the set of inner states of child Cl_B held by parent Cl_A must be the same as set of inner states held by Cl_B, which captures the commuting of Fig. 1.14 and says that the child cannot be more globally constrained than the parent. □

Definition 25 (Relaxed BI Refinement)**.** Let $Cl_A, Cl_C : Cl$ (def 6) such that Cl_C **inh** Cl_A (def. 11). Class Cl_C is BI conformant with Cl_A ($Cl_C \sqsupseteq_{BI} Cl_A$) with relaxed applicability and freeness when the following holds:

$$vos = vopsids\ Cl_A \lhd ((ops \circ ity)\, Cl_A)$$

$$Cl_C \sqsupseteq_{BI} Cl_A \Leftrightarrow ity\ Cl_C \sqsupseteq_{Ext} ity\ Cl_A \bowtie vos \wedge Cl_A\ \mathbf{Inhfree}\ Cl_C$$

Above, *vos* holds virtual operations of Cl_A; **Inhfree** is as per def. 23. □

References

1. Abrial, J.R., Cansell, D., Méry, D.: Refinement and reachability in Event_B. In: ZB2005, *LNCS*, vol. 3455, pp. 222–241. Springer (2005). DOI 10.1007/11415787_14
2. Amálio, N.: Generative frameworks for rigorous model-driven development. Ph.D. thesis, Dept. Computer Science, Univ. of York (2007)
3. Amálio, N.: Relaxing behavioural inheritance. In: Refine 2013, *EPTCS*, vol. 115, pp. 68–83 (2013)
4. Amálio, N.: Behavioural inheritance with relaxed but safe constraints grounded on data refinement. techreport, Birmingham City University (2018). URL `http://bit.ly/BI5nl0I`
5. Amálio, N.: Isabelle proofs of behavioural inheritance and relaxations. available online (2018). URL `http://bit.ly/2o0KaI2`
6. Amálio, N., Glodt, C.: A tool for visual and formal modelling of software designs. Sci. Comput. Program. **98, Part 1**, 52 – 79 (2015). DOI 10.1016/j.scico.2014.05.002
7. Amálio, N., Glodt, C., Kelsen, P.: Building VCL models and automatically generating Z specifications from them. In: FM 2011, *LNCS*, vol. 6664, pp. 149–153. Springer (2011)
8. Amálio, N., Kelsen, P.: Modular design by contract visually and formally using VCL. In: VL/HCC 2010, pp. 227–234. IEEE (2010). DOI 10.1109/VLHCC.2010.39
9. Amálio, N., Kelsen, P., Ma, Q., Glodt, C.: Using VCL as an aspect-oriented approach to requirements modelling. TAOSD **VII**, 151–199 (2010)
10. Amálio, N., Polack, F., Stepney, S.: An object-oriented structuring for Z based on views. In: ZB 2005, *LNCS*, vol. 3455, pp. 262–278. Springer (2005)
11. Amálio, N., Polack, F., Stepney, S.: UML+Z: Augmenting UML with Z. In: H. Abrias, M. Frappier (eds.) Software Specification Methods. ISTE (2006)
12. Amálio, N., Polack, F., Stepney, S.: Frameworks based on templates for rigorous model-driven development. ENTCS **191**, 3–23 (2007)
13. Banach, R., Poppleton, M.: Retrenchment: An engineering variation on refinement. In: B'98, *LNCS*, vol. 1393, pp. 129–147. Springer (1998). DOI 10.1007/BFb0053358
14. Banach, R., Poppleton, M., Jeske, C., Stepney, S.: Engineering and theoretical underpinnings of retrenchment. Science of Computer Programming **67**(2–3), 301–329 (2007)
15. Chalin, P., Kiniry, J.R., Leavens, G.T., Poll, E.: Beyond assertions: advanced specification and verification with JML and ESC/Java2. In: F.S. de Boer, et al. (eds.) FMCO 2005, *LNCS*, vol. 4111, pp. 342–363. Springer (2006)
16. Derrick, J., Boiten, E.: Refinement in Z and Object-Z: foundations and advanced applications. Springer (2001)
17. Dhara, K.K., Leavens, G.T.: Forcing behavioural subtyping through specification inheritance. In: ICSE-18: 18th Int. Conf. on Software Engineering, pp. 258–267 (1996). Also published as TR # 95 − 20*c* Dept. Comp Science, Iowa State University.

18. Fischer, C., Wehrheim, H.: Behavioural subtyping relations for object-oriented formalisms. In: AMAST 2000, *LNCS*, vol. 1816, pp. 469–483. Springer (2000)
19. Hall, A.: Using Z as a specification calculus for object-oriented systems. In: A. Hoare, D. Bjørner, H. Langmaack (eds.) VDM '90, *LNCS*, vol. 428, pp. 290–318 (1990)
20. Hall, A.: Specifying and interpreting class hierarchies in Z. In: Z User Workshop, Workshops in Computing, pp. 120–138. Springer (1994)
21. Harel, D., Kupferman, O.: On object systems and behavioural inheritance. IEEE Transactions on Software Engineering **28**(9), 889–903 (2002)
22. He, J., Hoare, A., Sanders, J.W.: Data refinement refined. In: ESOP'86, *LNCS*, vol. 213, pp. 187–196. Springer (1986). DOI 10.1007/3-540-16442-1_14
23. Hoare, A.: Proof of correctness of data representations. Acta Informatica **1**(1), 271–281 (1972). DOI 10.1007/BF00289507
24. ISO: Information technology—Z formal specification notation—syntax, type system and semantics (2002). ISO/IEC 13568:2002, Int. Standard
25. Jackson, D.: Software Abstractions: logic, lanaguage, and analysis. MIT Press (2006)
26. Leavens, G.T.: Jml's rich, inherited specifications for behavioural subtypes. In: ICFEM 2006, vol. 4260, pp. 2–34. Springer (2006)
27. Leino, K.R.M., Müller, P.: Using the spec# language, methodology, and tools to write bug-free programs. In: Advanced Lectures on Software Engineering: LASER Summer School 2007/2008, pp. 91–139. Springer (2010)
28. Liskov, B., Wing, J.: A behavioral notion of subtyping. ACM Trans. Program. Lang. Syst. **16**(6), 1811–1841 (1994)
29. Lupton, P.J.: Promoting forward simulation. In: Z User Workshop, pp. 27–49. Springer (1990)
30. Mayr, E.: Biological classification: toward a synthesis of opposing methodologies. Science **214**(30) (1981)
31. Meyer, B.: Applying "design by contract". Computer **25**(10), 40–51 (1992)
32. Meyer, B.: Object-Oriented Software Construction. Prentice-Hall (1997)
33. Meyer, B.: Touch of class: learning to program well with Objects and Contracts. Springer (2009)
34. Smith, G.P.: The Object-Z Specification Language. Kluwer Academic Publishers (2000). DOI 10.1007/978-1-4615-5265-9
35. Stepney, S., Polack, F., Toyn, I.: Patterns to guide practical refactoring: examples targetting promotion in Z. In: ZB 2003, *LNCS*, vol. 2651, pp. 20–39. Springer (2003)
36. Tschannen, J., Furia, C.A., Nordio, M., Meyer, B.: Automatic verification of advanced object-oriented features: The autoproof approach. In: LASER 2012, *LNCS*, vol. 7682, pp. 133–155 (2012)
37. Wehrheim, H.: Behavioral subtyping and property preservation. In: S.F. Smith, C.L. Talcott (eds.) FMOODS 2000, pp. 213–231. Kluwer (2000)
38. Wirth, N.: Program development by stepwise refinement. Communications of the ACM **14**(4), 221–227 (1971)
39. Woodcock, J., Davies, J.: Using Z: Specification, Refinement, and Proof. Prentice-Hall (1996)