Draft of 17 February 2003

Proving Pointer Program Properties

Part 1: Context and overview

Bertrand Meyer

ABSTRACT

Efforts to reason formally about programs, and in particular to *prove* their properties mathematically, have no practical value unless they can handle all the language facilities on which realistic programs depend. It is then not surprising that one of the biggest obstacles to the spread of such correctness-guaranteeing methods has been the lack of a good way to model the highly dynamic nature of the run-time structures created by object-oriented programs — and by most plain C or Pascal programs — with their heavy use of *pointers*, or *references*, from object to object.

The present discussion proposes a mathematical theory for modeling pointer-rich object structures and proving their properties.

The model only uses simple concepts from set theory: sets, relations, functions, composition, restriction, image. For run-time operations all it needs is the notion of *event*, a function yielding a new program state from an existing one.

The model has two principal applications:

- The *coarse-grained* version of the model, considering only the existence or not of a reference between an object and another, gives a basis for discussing overall properties of the object structure, defining as a result the correctness constraints of *memory management* and especially **garbage collection**, full or incremental. Mathematically, this model uses a binary *relation*.
- The *fine-grained* version, based on *functions* which together make up the relation of the coarse-grained version, integrates the properties of individual object fields. As a result, it allows **proving the correctness of classes** describing structures with luxurious pointer foliage, from linked lists and graphs to B-trees and double-ended queues.

This is part of a series of articles. See here for links to the others.

1.1 INTRODUCTION

Scope

Advances in the theory of programming enable us to reason more systematically about programs. This goal is not just academic any more, thanks in particular to recent tools that permit the mathematical development of significant industrial systems accompanied by a proof of their correctness.

A wider use of these techniques would be of great benefit to software technology, but obstacles remain, including *theoretical* obstacles. One of the most significant is the lack of a generally accepted mathematical model for the possibly complex structures, involving numerous *pointers* (or *references*) between objects, that we can direct our programs to create during their execution. We will study such a model and its application to a variety of problems, including:

- Memory management, especially garbage collection, for programs that create many objects linked by references.
- Mathematical proofs of properties of object-oriented components describing object structures involving possibly complex use of references.

Organization

This presentation includes four separate articles.

- The rest of Part 1, **Context and Overview**, explains the goals and assumptions of the project, then (1.2) describes its rationale by recalling the inevitability of using references when modeling systems. It also includes a summary of notations (1.3), a bibliography (1.4) and acknowledgments, all applicable to the full series.
- Part 2 presents a **coarse-grained model** that describes the pointer structure as a whole, ignoring individual object fields. The mathematical framework covers such concepts as abstract addresses, objects, links between objects, stack, heap, garbage, live objects, and such events as heap and stack allocation, full and incremental garbage collection, yielding a substantial set of theorems.
- Part 3 shows how to refine this first model into a **fine-grained model** for proving properties of individual classes and individual object fields.
- Part 4 applies the fine-grained model to **prove properties of specific classes** describing linked object structures. This part is not yet available at the time of writing, but the main results appear in a separate paper [12].

Trusted components

This work is part of a more general effort towards "Trusted Components" [10]: reusable software elements enjoying guaranteed properties. The project includes two complementary parts:

- A "low road" aimed at analyzing components built with current technology — library classes, COM objects, Enterprise Java Beans, .NET assemblies — through a *Component Quality Model* that has to take into account the state of the industry as it is.
- A "high road" aimed at producing components enjoying mathematically proved properties.

The present work is on the "high road"; it is a required milestone on that road, since as discussed in section 1.2 realistic components need pointers.

Why this should work

There have been numerous attempts before to provide a theory for pointer structures, none of which has established itself widely in practice — to the point that Abadi and Cardelli's treatise on the theory of objects [1] stays aways from pointers. It is fair to ask why the present one has a better chance. It rests its claims on three properties: simplicity; abstraction; and object-orientation.

First, the model exclusively relies on *simple mathematical concepts*. The mathematics involved is elementary set theory, nothing else. Anyone who understands sets, relations, union, intersection and the like can follow from beginning to end; no lambda calculus, higher-order logic, term rewriting, or calculi with names of various Greek letters.

Next, the model takes advantage of **abstract operators**. Transitive closure and other high-level operations on relations and functions yield simple and convincing results. The image operator, for example, is central to the developments below.

Third, we use the **concepts of object technology** and focus on properties of object-oriented programs.Most authors of theoretical work in the field define, as their basic problem, the need to model a non-object-oriented construct: the C++ or Java assignment to a "qualified target"

$$x.a := b$$

which requires modeling every operation as potentially affecting the full object store (in the same way that an array operation *a* [*i*] := *x* potentially affects any element of the array *a*). Yet O-O methodology implies that reference assignments must only occur within a routine of the corresponding class, under the unqualified form

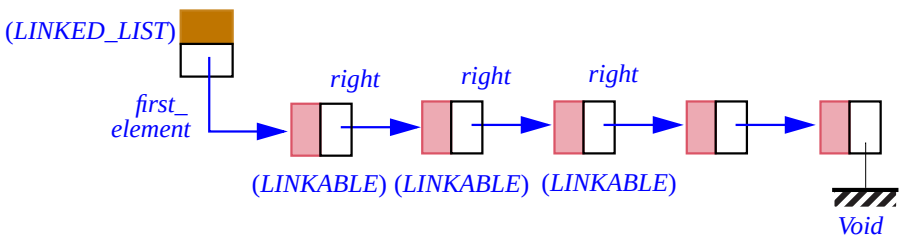
a := *b*

Like goto avoidance in elementary programming, this rule is both sound for program design and helpful for mathematical modeling.

We try to take advantage of object technology, whose *reusability* goal shifts the focus of proofs to **class invariants** expressing key properties of the pointer structures defined by a class. A typical example is

"Starting from the list header and following the reference *first_element* that leads to a list cell, and then the reference *right* that leads from a cell to the next, we will never hit the same cell twice, and eventually we will hit a *Void*"

as illustrated by this picture of a run-time linked list structure:



Run-time snapshot of a linked list

The class-based nature of object-oriented programs enables us to devote our efforts to proving that the corresponding reusable classes preserve these invariants. This seems the best way to avoid, in the words of Bornat [4], being

> *force*[*d*] *towards global reasoning* [*where*] *every* [*attribute*] *assignment seems to affect every assertion* [*about*] *the heap*

as seems hard to avoid if *x*.*a* := *b* is the object of our study. By contrast

> *The* [*classical*] *Floyd-Hoare treatment of assignment to a variable concentrates attention on assertions that involve that variable, leaving others untouched, making steps of local reasoning whose restriction to particular formulæ matches the locality of assignment.*

Proofs concentrating on classes give us back that locality. From such proofs — especially invariance proofs — we can infer the abstract properties of the routines of a class, as available to client code: Instead of *x.a := b*, a client application in genuine O-O programming uses calls of the form *x.r (b)*; the interface of the class, expressed in Eiffel by the contracts, gives the properties of such calls for all exported routines *r*.

This is indeed what we need to know when using pointer-manipulating programs. Programmers don't use pointer structures for their sake, but as convenient implementations of abstract structures having certain abstract properties. For example, a linked list as pictured above is useful as a representation of an abstract sequential structure; what is relevant, when you build and use such a structure through a reusable library class *LINKED_LIST*, is to know that a call

```
my_list.put_front (some_value)
```

results in a list having the same elements as before except for a new one, of content *some_value*, inserted at the front. This property is part of the contracts for the routine *put_right* of *LINKED_LIST* and the class as a whole:

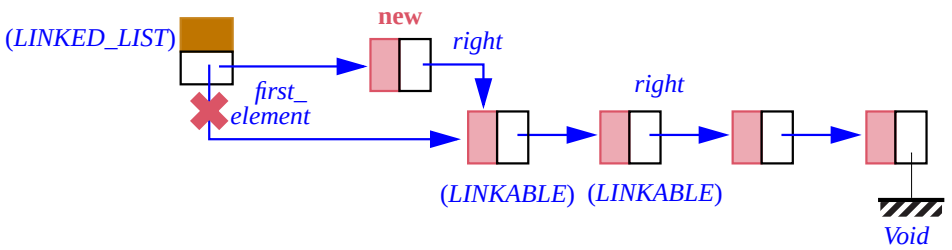
- The routine must ensure its postcondition, which will state that there is a new element at the front containing the desired value, and that the remaining elements have been preserved.

- The routine, like all others in the class, must preserve the invariant stated informally above (no cycle, *Void* at end).

It's in proving that *put_front* has these properties that we'll need a theory of pointer manipulations, but that theory can satisfy the principle of locality. Here the body of *put_front* might be of the form

```
create n
n.put_right (first_element)
first_element := n
```

which we may illustrate as



Inserting an element at the front

relying only on local reasoning about *first_element* and the contract of procedure *put_right* in class *LINKABLE*. To establish that the implementation of *put_right* satisfies this contract (stating that the routine reattaches the *right* link to the argument) uses reasoning that is in turn local to class *LINKABLE*.

Using operators on relations and functions

In expressing the properties that make up the contracts of pointer-rich structures, the abstract style of specification mentioned above will be particular effective. Authors who have already used such a style to discuss pointer programming issues include Möller [13] and Back *et al.* [3]; the present effort relies on their work and extends it.

Consider the property “We will never hit the same cell twice”, from the informal description of the linked list invariant. Abstract operators let us state it concisely as

$$(first\_element\ ;\ right^*) \cap id\ [LINKABLE] = \varnothing$$

to be understood as follows: *first_element* and *right* are functions, which for any object give another object, corresponding to the respective fields in *LINKED_LIST* and *LINKABLE*; the semicolon represents composition of functions or relations; the asterisk is reflexive transitive closure (the result of applying a function or relation zero or more times); and *id* [*X*] is the identity relation on a set *X*. So the formula states that the composition of *first_element* with any number of applications of *right* yields no identity pair (no pair of the form [*x*, *x*]) — the desired property. Without the abstract operators, one would have to write (see for example the style of [15]) something like

$$\forall\, x: \mathit{LINKED\_LIST} \bullet \forall\, m, n: \mathbb{N} \bullet m \neq n \Rightarrow \mathit{next}\,(x, m) \neq \mathit{next}\,(x, n)$$

(*Not the retained mathematical style.*)

with the auxiliary function definition

$$\begin{array}{lll} \mathit{next}\,(x, n) = & \mathit{first\_element}\,(x) & \text{if } n = 0 \\ = & \mathit{right}\,(\mathit{next}\,(x, n–1)) & \text{if } n > 0 \end{array}$$

The verbosity is striking; the added variables (x, m, n) play no useful role, and the recursive function definition is overkill. It's this kind of notational inflation that makes simple problems look complicated, leading to a a "big artillery" style of specification and proof that is hard to pursue very far in practice.

As another example, one of our theorems will state

→ *Coarse-grained model, page 7.*

$$[\text{T11}] \qquad \mathit{attached}\,(\bullet\, \mathit{Objects}\, \bullet) \subseteq \mathit{Objects}$$

where $r\,(\bullet\, X\, \bullet)$ denotes the image of a subset X under a relation r, that is to say, the set of elements to which r links an element of X; the relation *attached* links two objects if there is a reference from one to the other. The theorem, known as the "No Zombie" property, expresses the basic rule that following a reference from any object will lead another object: there are no dangling references. It's an invariant that all operations on references must be preserve. Without the image operator, this could be expressed as

$$\forall\, x: \mathit{Objects} \bullet \forall\, l: \mathit{Attributes}\,(\mathit{Class}\,(x)) \bullet (\exists\, y: \mathbb{N} \bullet \exists\, l: \mathit{Links} \bullet \mathit{reference}\,(x, l) = y) \Rightarrow y \in \mathit{Objects}$$

It's not hard to decide which version to show to a human reader or an automatic prover.

Proof support

A rule applied to this effort is to ensure that all stated properties are proved, not just manually but mechanically. At the time of writing the mechanical part is in progress, using the Atelier B formal development workbench [5].

Mathematical basis and style

The mathematical concepts used are from elementary set theory. The notations are introduced on first use and summarized at the end of Part 1.

→ *"CONVENTIONS AND NOTATIONS", page 14.*

"Events" follow the ideas of B [2]. Syntactically, they are expressed in an Eiffel-like form that should be immediately understandable, specifying for each event its condition of applicability (precondition: **require**), its **operation** on the state (body: **do**), and properties of the resulting state (postcondition: **ensure**), which must be proved as theorems. The model also includes **invariants**, which every event must be proved to maintain.

The reader may find the style of mathematical development too detailed, perhaps tedious, with the inclusion of many theorems, some seemingly obvious. The aim has been to help understand the issues in depth and to prepare for machine-checked proofs using Atelier B.

1.2 RATIONALE: WHY USE REFERENCE-BASED MODELS

To prepare the model's presentation in the next articles of the series, we first review why pointers are useful and why they cause trouble. This survey (which you can peruse quickly if you are already convinced of the inevitability of pointers, and of the theoretical problems they raise, such as dynamic aliasing) sets the precise context of the rest of the discussion.

The lure of dynamic models

Programmers today want to take advantage of the power of object technology to build class structures whose run-time instantiations will consist of large numbers of objects containing many references to each other. It would be wrong to dismiss the resulting complexity as self-inflicted: unlike the complexity of a programming language or an analysis notation, which can be criticized as the result of poor design, the complexity faced here results from our attempts to model (using a language or notation that may be beyond reproach) inherently complex aspects of the world. An employee belongs to a department, a department has a budget, a budget has an applicable period, a period has a beginning date and an ending date, and so on. Criticizing object models because they lead to objects that may include many references fields — a reference field of type *DEPARTMENT* in class *EMPLOYEE* etc. — would be all the more unfair that one of the achievements of object technology is precisely that it enables us to describe *object* structures of great apparent complexity while retaining simplicity in the design of the corresponding *classes* and their mutual relations, that is to say, the software model.

Another property of reference-rich object structures is that they usually take advantage of dynamic allocation, meaning that execution creates objects on demand. You do not need to know in advance how many *EMPLOYEE* objects your program will need: the program will create a "list of employees" object at the beginning of its execution; if later on it needs a new *EMPLOYEE* object it will create it at that time, for example from an employee record in a database, and add to the list a reference to that object. This is both more convenient and more economical than having to plan a maximum number of *EMPLOYEE* objects, with the risk of failing in the occasional case that requires more of them than planned, and of wasting memory in all others.

An added advantage is — in a good object-oriented environment — the ability to rely on *automatic memory management* mechanisms for the allocation and de-allocation of memory for objects, and particularly on an automatic *garbage collection* facility for recycling the memory space used by objects that can provably be of no longer use to the current execution of a program. Doing such recycling manually in the program requires considerable programming work of no direct relevance to the application and, even more worryingly, can easily lead to grave errors, hard to debug, in the case of mistakenly recycling space for a still active objects. If the programming language provides references within a suitable type system, the programming environment can, through its garbage collector, remove all such worries.

The alternative to using references would be, as in the days before object technology — indeed, the days before Pascal —, to model everything through integers and arrays. But no one who has had a taste of object-oriented analysis, design or programming will voluntarily go back and renounce the benefits of a model supporting references, dynamic object creation, and automatic garbage collection.

The challenges of reasoning with references

Unfortunately, approaches to mathematics-based software development and proof still generally limit themselves to dealing with basic data types such as integers and booleans, lacking a good formal model for object structures involving references. The reason is that references introduce properties that complicate the mathematical description of object structures.

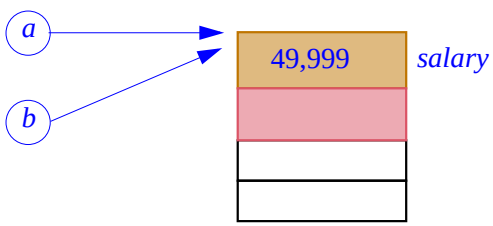
The most significant is **dynamic aliasing**, the ability for an object to become known under two different names, because two different references point to it. Dynamic aliasing breaks some common, widely assumed modes of reasoning. We are used to assuming that in a situation such as the following we understand what is going on:

```
-- Before: SOME_PROPERTY holds of a
Apply SOME_OPERATION to b.
-- After: SOME_PROPERTY still holds of a
```

We assume, at the position marked Before, that a certain property holds of some thing we call *a*. Then someone applies an operation to another thing, *b*. This operation does not involve *a* in any explicit way deducible from the description of the operation itself. Then we expect that after the operation, the property we relied on still holds of *a*, since the operation should not have "touched" *a*.

With the introduction of references, unfortunately, this simple mode of reasoning does not necessarily hold. Assume that *a* and *b* denote references. A property of a reference is that it may refer to an object; using Eiffel's terminology we'll say that a reference may be **attached** to an object. (A reference that is not attached to any object is said to be **void**.) Then *a* and *b* might be attached to the same object, and an operation on *b* may invalidate a property of *a*, if that property characterizes the attached object.

For example, in the figure below, both *a* and *b* are attached to an *EMPLOYEE* object with several fields, including *salary*. If *SOME_PROPERTY* says that the monthly salary of the object attached to *a* is less than fifty thousand euros, and *SOME_OPERATION* increases by one the salary of the object attached to *b*, *SOME_PROPERTY* will not hold at the point marked After above.



Aliasing

This is a case of **aliasing** (an object being known through two different names, here *a* and *b*). With non-reference variables, belonging to types that we will call *expanded* using Eiffel terminology again, no such problem arises: if *a* and *b* are different variables of type *INTEGER*, an expanded type, an operation on *b* may never change a property of *a*.

What makes aliasing even more tricky is that in object-oriented languages — and any language permitting pointers, such as Pascal or C — aliasing can be *dynamic*: we cannot foresee all cases before run time, since aliasing may result from a reference assignment

```
b := a
```

which may be executed, or not, depending on input data or interactive user actions; so in general we have no easy way, when looking at two reference names *a* and *b* in the program text, to know whether or not they may during execution ever become attached to a common object. This makes reasoning about programs much more difficult than with programming models where all types are expanded.

As already noted, we cannot blame such complexity on programming models alone. The problem is more fundamental, following from our ability to refer to things of the world around us in more than one way. Reading a "Before" assumption and an operation in the above style

```
-- I heard that both of the CEO's in-laws make less than 50K.
Memo to personnel: raise Jill's salary by one euro
```

we might hastily deduce that after the operation the original assumption still holds, but we would be wrong if it turns out that Jill's son is married to the CEO. Such aliasing can, in life as in computer programs, be dynamic, since the marriage may take place between the time we hear the gossip and the time Jill's salary is raised.

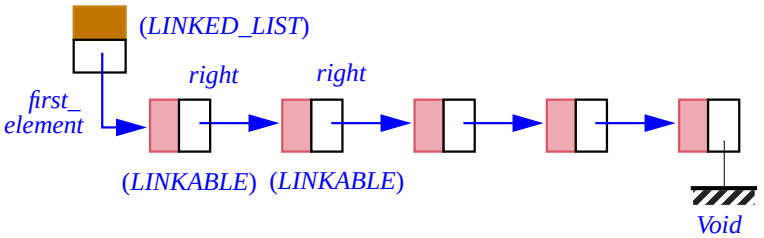
> Such examples suggest that it may be unfair to blame pointers for difficulties which can be traced to the human ability to call things by more than one name. "The beautiful daughter of Leda", "Menelas's wife" and "Helen of Troy" all denote the same person; readers of Saint-Simon are supposed to know that "Monsieur" is the King's brother; and more than one amateur of Russian novels has had trouble remembering on page 467 that "Daria Alexandrovna" was introduced as the "Countess Oblonsky" on page 5 and called "Dounia" on page 35, but elsewhere was just "Dolly".

Properties of object structures

Let us get a first idea of where we are heading by considering the kind of properties that — assuming we overcome the difficulties just reviewed — we want to be able to prove. Here is a typical example; simple, but beyond the realm of many proof systems. The example, already previewed, is class *LINKED_LIST* from the EiffelBase library, reduced to its essentials. Using this class leads to run-time object structures that we may picture as follows:

→ *Page 4.*

Run-time snapshot of a linked list



This involves two classes: *LINKED_LIST* proper, representing the list header (object at the top left); and *LINKABLE*, representing the list elements and meant to be used only by *LINKED_LIST*, not directly by clients of *LINKED_LIST*. The shaded cells represent information associated with the list, such as the number of items (in the *LINKED_LIST* object) and actual list element contents in the *LINKABLE* objects.

Such structures are the result of class declarations of the form

```
class LINKED_LIST [G] feature
        first_element: LINKABLE [G]
        ... Routines (see below) ...
end -- class LINKED_LIST
```

```
class LINKABLE [G] feature
        right: LINKABLE [G]
        ... Routines (see below) ...
end -- class LINKABLE
```

The kinds of properties we will want to prove include:

- No cycles: if at any time during the life of such a list we follow the *first_element* reference from a *LINKED_LIST* object, and then follow the sequence of *right* references in *LINKABLE* objects for as long as applicable, we will never encounter the same *LINKABLE* object twice.
- Void termination: any such sequence eventually links to *Void*.
- No tail sharing: starting from two different *LINKED_LIST* objects, such sequences of *LINKABLE* objects are disjoint.
- The procedure *put_front* (given next) will result in a list with one more item than before.
- The procedure *remove_front* will result in a list with one fewer item, and cause the addition of one object to the list of "garbage", that is to say, objects whose memory may be safely returned to the operating system.

The first two of these properties are invariants; they must be ensured by all the initialization procedures of the class ("constructors" in C++ terminology), and preserved by all exported routines such as *put_front* and *remove_front*.

Here are these two routines — with Eiffel assertions omitted although they will figure in any acceptable version of the classes. First *put_front* in *LINKED_LIST*:

```
put_front is
        -- Add element to beginning of list.
    local
        n: LINKABLE [G]
    do
        create n
        n.put_right (first_element)
        first_element := n
    end
```

In a practically useful library, *put_front* would have an argument of type *G* representing the value to be stored in the new element; in this discussion, however, we may omit the argument since we focus on the references between objects (the arrows on the last figure) and ignore any non-reference fields in objects (the figure's shaded areas).

Procedure *put_front* calls the following procedure from class *LINKABLE* (available only, thanks to the mechanism of selective export, to class *LINKED_LIST* and its descendants):

```
put_right (other: LINKABLE [G]) is
        -- Make other the new right neighbor of current object.
    do
        right := other
    end
```

Procedure *remove_front* in *LINKED_LIST* will be written:

```
remove_front is
        -- Remove first element of list.
    require
        not_empty: first_element /= Void
    do
        first_element := first_element.right
    end
```

We will explore how to prove the properties of such routines.

1.3 CONVENTIONS AND NOTATIONS

This section serves as reference for the whole set of articles.

The model consists, in the B style, of constants, variables, and events that affect the variables. Constants and variables are sets (including the special cases of relations and functions).

In addition to events, the formal elements are of four kinds: *axioms* introducing new elements (which may themselves be sets) of known sets; *definitions* introducing new elements in terms of previously introduced ones; *invariants* expressing properties that we must prove to be preserved by every event; and *theorems* that we must prove to follow from the other properties. All these elements are numbered in a single sequence, with numbers respectively starting with A, D, I and T.

For example the first axiom is [A1]*, the first definition* [D9]*, the first invariant* [I3] *and the first theorem* [T7].

We will try to keep our model minimal by limiting the number of elements that we explicitly postulate: axioms and invariants. (Definitions and theorems do not introduce any new assumption.) To help keep track of this goal, every new axiom or invariant is marked by a ● in the right margin.

The rest of this section gathers the notations used in the remaining articles. It is not necessary to study it on first reading since all non-elementary notations are introduced on first use. Sources for the notation include Z, B, and reference [7].

Naming conventions

Symbol	Name	Typical use	Meaning
A, B, C, …	Set naming convention	*A*	(Names of sets, especially of addresses or objects, but not relations or functions, usually start with an upper-case letter.)
a, b, c, … *r,* … *f,* …	Member, relation, function naming convention	*a*	(Names of set elements, functions and relations usually start with a lower- case letter.)

General

Symbol	Name	Typical use	Meaning
$\triangleq$	Definition	$x \triangleq E$	From now on, understand *x* as an abbreviation for *E*.
:	Distinguished member	*a*: *S*	From now on, assume that *S* (a known set) has a specific member, to be called *a*.

Properties of the model

Symbol	Name	Typical use	Meaning
A*i*	Axiom number *i*.	[A1] *a*: *S*	Assume that *S* (a known set) has a specific member, to be called *a*.
D*i*	Definition number *i*. (Must be of one of the two forms shown on the right.)	[D2] $x \triangleq E$	From now on, understand *x* as an abbreviation for *E*.

Symbol	Name	Typical use	Meaning
I*i*	Invariant number *i*.	[I3] $A = B$	All events must preserve the given property (proof obligation).
T*i*	Theorem number *i*.	[T4] $A = B$	The given property follows from previous definitions, axioms and theorems

Sets

Symbol	Name	Typical use	Meaning
$\in$	Set membership	$a \in S$	(True or false.) a is a member of S.
$\varnothing$	Empty subset	$\varnothing$	Subset (of any set) that has no members.
$\subseteq$	Set inclusion	$X \subseteq Y$	(True or false.) Every member of X (if any) is also a member of Y.
$\cup$	Union of subsets	$A \cup B$	Union of A and B: the subset whose members are the members of either or both of A and B.
$\cap$	Intersection of subsets	$A \cap B$	Intersection of A and B: the subset whose members are the members of both of A and B.
–	Difference of subsets	$B - A$	Difference of B and A: the subset whose members are the members of B that are not members of A.
———	Complement of a subset	$\overline{X}$	Complement of X: the set whose members are members of the enclosing set that are not members of X.
$\times$	Cartesian product	$A \times B$	Cartesian product of A and B: the set of pairs $[a, b]$ where a is a member of A and b a member of B.
{ }	Extension (enumeration)	$\{a, b, c\}$	The set whose members are a, b and c.

Symbol	Name	Typical use	Meaning
$\mathbb{P}$	Powerset	$\mathbb{P}\ (X)$	The set whose members are the subsets of X.
$\mathbb{F}$	Finite powerset	$\mathbb{F}\ (X)$	The set whose members are the finite subsets of X.
$\mathbb{N}$	Set of integers	$\mathbb{N}$	The set whose members are all positive integers
$\mathbb{N}^{\bullet}$	Set of positive integers	$\mathbb{N}^{\bullet}$	The set whose members are all positive integers

Logic

Symbol	Name	Typical use	Meaning
$\neg$	Negation	$\neg P$	True if and only if P is false.
$\Rightarrow$	Implication	$P \Rightarrow Q$	True unless P is true and Q is false.
$\forall$	Universal quantifier	$\forall\, j\colon A \bullet P$	True if and only if P holds of every x that is an element of A. (True if A is an empty subset.)
$\exists$	Existential quantifier	$\exists\, j\colon A \bullet P$	True if and only if P holds of at least one x that is an element of A. (False if A is an empty subset.)

Relations and functions

Symbol	Name	Typical use	Meaning
$[\ \ ]$	Pair	$[a, b]$	The pair whose first element is a and second element is b.
	Relation		A relation of source set A and target set B is a set of pairs whose first element is a member of A and second element a member of B.

Symbol	Name	Typical use	Meaning
$\leftrightarrow$	Set of relations	$A \leftrightarrow B$	The set of relations of source set A and target set B, i.e. the set of pairs $[a, b]$ such that a is a member of A and b a member of B.
$⇼$	Set of finite relations	$A ⇼ B$	The set of relations of source set A and target set B that are finite sets. (Their domain and range are both finite.)
$⇸$	Set of functions (possibly partial)	$A ⇸ B$	Functions of source set A and target set B: members of $A \leftrightarrow B$ which for any member a of A have at most one member pair whose first element is a.
$⇻$	Set of finite functions	$A ⇻ B$	Functions of source set A and target set B, whose domain is a finite subset
$\rightarrow$	Set of total functions	$A \rightarrow B$	Total functions of source set A and target set B: members of $A ⇸ B$ whose domain is A. (For any member a of A, such a function has exactly one member pair whose first element is a.)
domain	Domain	**domain** (r)	For a relation r of source set A, the subset of A whose members are all a such that r contains a member pair whose first element is a.
range	Range	**range** (r)	For a relation r of target set B, the subset of B whose members are all b such that r contains a member pair whose second element is b.
$r \backslash X$	Restriction	$r \backslash X$	Restriction of relation r to a subset X of its target set: the relation containing every pair in r whose first element is a member of X.
id	Identity relation	$id\ [X]$	Identity relation on X: the relation of source set X and target set X whose members are all the pairs having the same first and second element.

Symbol	Name	Typical use	Meaning
(Exponent)	Iteration of a relation	r^n	(For $n \geq 0$.) r iterated n times: if $n = 0$, the identity relation; if $n > 0$, r composed, recursively, with r^{n-1}.
$^+$	Transitive closure of a relation	r^+	Transitive closure of r: $r^1 \cup r^2 \cup r^3 \cup \ldots$
*	Reflexive transitive closure of a relation	r^*	Reflexive transitive closure of r: $r^0 \cup r^+$
cyclic	Cyclic relation	*cyclic* (r)	(True or false.) r is a cyclic relation: among the members of r are n pairs $[a_1, a_2]$, $[a_2, a_3]$, …, $[a_{n-1}, a_n]$, $[a_n, a_1]$ for some $n \geq 1$. (True if r is an identity relation, or contains an identity relation, e.g. if it is a transitive closure.)
$^{-1}$	Inverse of a relation	r^{-1}	Inverse relation of r: the set of pairs $[a, b]$ such that the pair $[a, b]$ is a member of r. The inverse of a function is a relation, not necessarily a function.
( )	Function application	$f(x)$	Application of a function f to a member a of its domain: the element b (there is exactly one) such that $[a, b]$ is a member of f.
(• •)	Image by a relation	$r(\bullet A \bullet)$	Image of subset A under relation f: the set of all b (if any) such that r has a pair $[a, b]$ for some a that is a member of A.
(• •)	Image by a function	$f(\bullet A \bullet)$	Image of subset A under function f: the set of all b (if any) such that $b = f(a)$ for some a that is a member of A. (Definition equivalent to previous one but applied to case of a function.)

Symbol	Name	Typical use	Meaning
curry	Currying	*curry* (*f*)	Curried version of *f* (*f* specialized on its first argument): if *f* is a member of $A \times B \to C$, *curry* (*f*) is a member *g* of $A \to (B \to C)$ such that, for any *a*, *g* (*a*) is the function *h* such that, for any *b*, *h* (*b*) = *f* (*a*, *b*). This definition uses total functions but immediately generalizes to partial functions and arbitrary relations.

Events and assertions

Symbol	Name	Typical use	Meaning
require	Precondition	**require** *condition*	Event may occur if and only if *condition* is satisfied.
ensure	Postcondition	**require** *condition*	Theorem (must be proved): Event will yield a result satisfying *condition*.
||	Parallel execution	*x* := *a* || *y* := *b*	The two operations occur in parallel.

1.4 REFERENCES

This bibliography applies to the entire series of articles.

[1] Martín Abadi and Luca Cardelli: *A Theory of Objects*, Monographs in Computer Science, Springer-Verlag, 1996.

[2] Jean-Raymond Abrial, *The B Book*, Cambridge University Press, 1995.

[3] Ralph Back, X. Fan and Viorel Preoteasa: *Reasoning about Pointers in Refinement Calculus*, Technical Report, Turku Centre for Computer Science, Turku (Finland), 22 August 2002.

[4] Richard Bornat: *Proving Pointer Programs in Hoare Logic*, in *Mathematics of Program Construction*, Springer-Verlag, 2000, pages 102-106.

[5] ClearSy [name of company, no author listed]: Web documents on Atelier B, www.atelierb.societe.com, last consulted December 2002.

[6] C.A.R. Hoare and He Jifeng: *A Trace Model for Pointers*, in *ECOOP '99 — Object-Oriented Programming*, Proceedings of 13th European Conference on Object-Oriented Programming, Lisbon, June 1999, ed. Rachid Guerraoui, Lecture Notes in Computer Science 1628, Springer-Verlag, pages 1-17.

[7] Bertrand Meyer: *Introduction to the Theory of Programming Languages*, Prentice Hall, 1990.

[8] Bertrand Meyer: *Object-Oriented Software Construction, 2nd edition*, Prentice Hall, 1997.

[9] Bertrand Meyer, Christine Mingins and Heinz Schmidt: P*roviding Trusted Components to the Industry*, in *Computer* (IEEE), vol. 31, no. 5, May 1998, pages 104-105.

[10] Bertrand Meyer et al.: Trusted Components papers at se.inf.ethz.ch, last consulted December 2002.

[11] Bertrand Meyer: *A Framework for Proving Contract-Equipped Classes*, to appear in *Abstract State Machines 2003 - Advances in Theory and Applications*, Proc. 10th International Workshop, Taormina, Italy, March 3-7, 2003, eds. Egon Boerger, Angelo Gargantini, Elvinia Riccobene, Springer-Verlag 2003. Pre-publication copy at www.inf.ethz.ch/~meyer/publications/, last consulted January 2003.

[12] Bertrand Meyer: *Towards Practical Proofs of Class Correctness*, at se.inf.ethz.ch/ongoing/references, last consulted February 2002.

[13] Bernhard Möller: *Calculating with Pointer Structures*, in *Algorithmic Languages and Calculi*, Proceedings of IFIP TC2/WG2.1 Working Conference, Le Bischenberg (France), February 1997, Chapman and Hall, 1997, pages 24-48.

[14] Joseph M. Morris, *A general axiom of assignment; Assignment and linked data structures*; *A proof of the Schorr-Waite algorithm*. In *Theoretical Foundations of Programming Methodology*, Proceedings of the 1981 Marktoberdorf Summer School, eds. Manfred Broy and Gunther Schmidt, Reidel 1982, pages 25-51.

[15] John C. Reynolds: *Separation Logic: A Logic for Shared mutable Data Structures*, in Proceedings of 17th Annual IEEE Symposium on Logic in Computer Science, Copenhagen, July 22-25 2002.

[16] Norihisha Suzuki, *Analysis of Pointer "Rotation"*, in *Communications of the ACM*, vol. 25, no. 5, May 1982, pages 330-335.

Acknowledgments

(This section expresses thanks for feedback received, without any implication of endorsement or agreement.) Earlier versions of the approach discussed in this series of articles have presented in several forums, leading to important comments and criticism: at an IFIP WG 2.3 meeting (Tony Hoare, John Reynolds, Natarajan Shankar, Michel Sintzoff); at the Turku Centre for Computer Science (Ralph Back, Viorel Preoteasa); at the Lipari Summer School of August 2002 (Jean-Raymond Abrial — also at an ETH presentation and email correspondence — and Egon Boerger); at Monash University (Christine Mingins); at the Formal Approaches To Software (FATS) seminar at ETH (Robert Staerk). Karine Arnout provided detailed comments on an earlier version of the paper. The work on proving classes is pursued with Bernd Schoeller, who made important suggestions on part 2.