A Z Patterns Catalogue : I

specification and refactorings, v0.1

Susan Stepney, Fiona Polack, and Ian Toyn

University of York Technical Report YCS-2003-349

January 2003

Contents

Preface

The various volumes in the "Z Patterns Catalogue" series of reports, outlined below, are evolving documents – as we discover and are informed of more patterns, we will add them to new versions of the reports.

The three reports, history and plans for the future

- **I : specification and refactoring** (Stepney, Polack, Toyn)
  - v0.1 – Jan 2003 – The initial structure, with a focus on promotion as a generative pattern, and refactoring, with many skeleton patterns (particulalry in the developmental section)
  - v0.2 – fleshed out skeletons, more patterns, and material from *Z in Practice*
- **II : definition and laws** (Valentine, Stepney, Toyn)
  - v0.1 – mid 2003 – The initial structure, of a rich mathematical toolkit
  - v0.2 – More generic patterns, including a type-constrained generic schema toolkit, and patterns for generating toolkits by abstraction
- **III : proof and refinement** (Cooper, Stepney, Woodcock)
  - v0.1 – end 2003 – The initial structure, with proofs of intereseting properties, and refinement as a generative proof pattern
  - v0.2 – Refactoring proofs, retrenchment as "approximate proof refactoring"

Chapter 1

Introduction

1.1 Background

Formal methods have been used in computer systems development for decades. The binary logic of hardware circuits can be designed and analysed using well-understood mathematical approaches. Software can be characterised in various ways that are amenable to formalisation. For example

- mappings between states can be represented as mathematical relations
- sets of data, to be selected from, merged, updated and compared, can be represented in set theory

Some formalisms, particularly the most mature forms used for hardware design, are compact, well-defined, and well integrated in the development process: they are specialised methods (or tools) for specialist developers. However, most software-oriented formalisms are under-exploited in commercial-scale development, because they are

- not properly integrated in existing development processes
- poorly supported by development tools

Whereas in mature development approaches, the stages and steps of development conform to clear and generally-accepted patterns, in these immature areas, there is more art than science; development success depends more on the character and skills of personnel than on the power of the methods. Notations, methods and tools for formal software specification and development require specialist knowledge, in an area that is not generally recognised as meriting any development specialism.

This report represents a contribution to the commercial acceptance of formality, specifically of the Z notation. Z[1] is a mathematical notation for typed set theory, with some syntactic sugar and built-in operator support for a certain kind of tuple constructor, the schema. It is a powerful *notation*, with few inbuilt assumptions about any design philosophy or development *method* of its own. This power and freedom can make it hard for the newcomer to decide how to structure and develop a Z specification, and hard for a reviewer or implementor to comprehend a specification written in an unfamiliar style. Our motivation is to make Z more usable by commercial non-specialist developers. No new Z is presented (either Z theory or usage, or indeed Z illustrations). However, the concepts of *pattern* and *refactoring* are applied to enhance the "semantic structure" of Z, thereby helping the writing, reading and presentation of Z.

This report assumes at least a familiarity with set and predicate notations, and of the specialised symbols used in Z. However, deep knowledge or expertise in writing or reading Z is not required.

The central premise of the work is that formalisms such as Z have a role to play in general software development. The patterns and refactorings should enable

- writing of formal texts by generalists, because the patterns present formal solutions to common problems
- development of tools to support the use of formal methods by generalists, by recognising and assisting in the application of patterns, and by breaking down the formal concepts into mechanisable or tool-supportable components

1.2 Motivation

Our motivation for investigating patterns and refactoring in the context of the Z formal notation comes from experience in the industrial use of Z, and a desire to make Z more usable by commercial non-specialist developers. One of the authors (SS) was a member of Logica UK's Formal Methods Team (LFM), where she worked extensively on large-scale commercial specification and proof, including a compiler [Stepney & Nabney 2003]; an electronic Purse [Stepney *et al.* 2000]; and a Smart Card Operating System [Stepney & Cooper 2003]. [Stepney 1998] reports

[1] We refer to the two main variants of Z thus: as ZRM, for that given in [Spivey 1992b]; as ISO-Z, for that given in [ISO-Z 2002]. By 'Mathematical Toolkit', we mean those well-known Z definitions given in [Spivey 1992b, chapter 4] and [ISO-Z 2002, annex B].

on issues to do with performing proofs on these industrial-scale Z specifications, and sketches requirements for proof tool support to help in this task.

Even when starting a specification and proof task from scratch, a commercial development rarely starts from a clean sheet of paper. Implementation details often constrain what can be done, and how the specification can be structured. When enhancing an existing specification, these constraints are even more important.

1.2.1 Impact analysis

Often a customer decides on an upgrade and requires this to be added to the specification. In simple cases this merely requires adding an operation (assuming a state and operations style specification) to capture the new functionality. In more complex cases it may also require the adding of state. More complex cases may require radical alterations to the specification, because the change subverts a modelling assumption of the original.

Part of the maintenance process is *impact analysis*: determining the cost and consequences of a proposed change before making that change. The effect on the formal specification and proof should be included in this impact analysis, so that the customer can realise the actual effect of what looks to them to be a simple implementation change.

For example, in one of LFM's projects, a customer added an operation with a different execution style. This new operation used a *push* model (the system waiting for a command, then responding to whatever command arrived), whilst all the other operations used a *pull* model (the system actively looping until it finished processing a sequence of commands). The change of model had little impact on the implementation, requiring only a simple flag local to the new operation. The impact on the specification was somewhat larger. The specification had to be revised to include a flag as a global state component. Every existing operation specification had to be changed, so that the flag was explicitly switched *off*. We also had to provide some subtle argumentation as to how this global specification variable corresponded to the implementation's local variable.

The impact of this change was significant in a hand-crafted specification; in many developments, this sort of problem is a deterrent to the use of formal methods.

1.2.2 Maintaining conventions

It is essential during maintenance to keep the formal and non-formal parts of the specification up to date. Indexing, commenting, and layout conventions all need to be maintained. This can be difficult if the conventions have not been documented and are not tool-supported. In LFM's projects, there was a degree of continuity of the staff, but the conventions had to be explained to new staff. Consequently, we started to describe some simple Patterns for these conventions, to aid the original specification process, and to help with maintenance.

In this report, patterns are considered first (chapters 2 to 11). Then, refactoring is explored (chapters 13 to 17). The report concludes with some discussion of ways forward and possible tool support (chapter 18 onwards).

Part I

Patterns

Chapter 2

Patterns

2.1 Background

Patterns [Alexander *et al.* 1977] capture the knowledge about the use of a language, acquired by experts or experienced users of the language, and express it in ways that (should) assist less experienced users to construct complex expressions in the language.

Patterns have been introduced into software engineering, to document and promote best practice, and to share expertise. A pattern provides *a solution to a problem in a context*. Existing patterns cover a wide range of issues, from coding standards [Beck 1997], through program design [Gamma *et al.* 1995], to domain analysis [Fowler 1997], and meta-concerns such as team structures and project management [Coplien 1995]. The pattern concept has been extended with antipatterns, illustrating developmental pitfalls and their avoidance or recovery [Brown *et al.* 1998].

Patterns do not stand in isolation. As Alexander, the inventor of the concept, explains [Alexander *et al.* 1977], a *Pattern Language* is a collection of patterns, expressed at different levels, that can be used together to give a structure at *all* levels to the system under development. The names of the patterns provide a *vocabulary* for describing problems and design solutions. (In this sense, a pattern language is simply a re-expression of an introductory text on a language.)

Typically, a pattern comprises a template or algorithm and a statement of its range of applicability. A catalogue records pattern descriptions, organised to facilitate pattern selection. For example, [Gamma *et al.* 1995] catalogue *creational patterns*, those providing architectural guidance; *structural patterns*, those guiding component construction; and *behavioural patterns*, which relate to the interaction of components. In providing for the selection of appropriate patterns, the description of the **intent** of the pattern is crucial. This describes the situation for which the

pattern is appropriate.

The pattern catalogue uses meaningful pattern names to guide users to appropriate patterns. It is also common to use a visual representation. For instance, [Gamma *et al.* 1995] and [Larman 2001] use UML diagrams to visualise object-oriented program and design patterns. [Gamma *et al.* 1995] assist the developer by providing connectivity diagrams to show how, for example, use of particular creational patterns suggests particular structural and behavioural patterns. A good pattern catalogue can be applied to assist all elements of construction of a description (program, design etc) in its language – it is possible to construct a Smalltalk program using [Gamma *et al.* 1995]'s patterns.

Patterns are usually specific to the language for which they are written: [Gamma *et al.* 1995] note that some patterns provided for Smalltalk programming are built-in features of other object oriented programming languages. In the formal language context, some Z patterns for identifying proof obligations would be irrelevant in the tool-supported B Method, in which the corresponding proof obligations are automatically generated. Equally, if a Z practitioner has already decided to use the Delta/Xi style, then the Delta/Xi pattern is superfluous. Furthermore, if a different Z style is used, most of the patterns written for use with the Delta/Xi pattern (promotion, change part of the state etc) are irrelevant. However, other patterns generalise, or occur in similar forms across many media (eg across languages, development phases, contexts):

- all notations require commentary which is clear, consistent, and adds meaning to the text;
- all notations have common usage conventions that can be expressed as patterns.

2.2 Antipatterns

The concept of patterns in software engineering has been extended to *antipatterns* [Brown *et al.* 1998]. An antipattern presents an example of poor practice, a pit into which developers (etc) often fall, and a way of avoiding or mitigating the resultant effects.

In [Brown *et al.* 1998], most of the antipatterns, which relate to software configuration management, describe universally poor practice. However, in other contexts,

and particular in notations such as Z, one developer's antipattern is another's pattern. This is because a formal text can have many purposes: a pattern that is used to simplify the proof of formal conjectures may reduce the readability of the Z text.

In writing antipatterns (and indeed patterns), and in selecting patterns for application, it is therefore important to consider the purpose of the description. The patterns presented here are most appropriate when the primary purpose of the Z specification is for communication; we are also working on patterns for other purposes including refinement, implementation and proof.

2.3 Generative patterns

In object-oriented programming, generative patterns are an element of adaptive programming. Patterns written in a meta-language are used to automatically derive programs in the object-oriented language. This is analogous to conventional compilation of a high-level program into a lower-level program [Lopes & Lieberherr 1994].

A looser meaning of generative pattern in object-oriented patterns work is the application of a series of patterns to create a program. Note that this is quite different from the creational patterns of [Gamma *et al.* 1995]: the latter are patterns that can be used to create or refactor specific elements of a program (classes, structures, generic operations etc).

It is impossible to automatically generate a specification from a meta-language template. The process of (commercial) specification establishes the requirements and progressively assembles an abstract description of a suitable system to meet the requirements. There can be no safe meta-level for a description that is continually and actively evolving. However, the looser definition of generative patterns is clearly applicable.

Chapter 3

Patterns in Z

3.1 Motivation

Z textbooks introduce the mathematical basis of Z, the notation, and essential elements of the use of Z. However, few books provide advice on how to "do" Z in practice. Illustrations clearly show how a feature was used by the author, but context and intent are implicit, and there is rarely any advice on how to reuse or adapt the Z text.

The pattern language represents the experience we and others have accumulated in writing and maintaining specifications. For experienced formalists, it offers a language for describing or summarising the approaches used in a particular specification. For the less experienced developer, the patterns give guidance on ways of using Z to solve certain problems, all the way from small scale syntactic issues, right up to using Z as a valuable part of the entire development process.

There are currently only a small number of well-known conventions for using Z, and many users are unaware that other approaches are possible. For example, the Delta/Xi style ("state and operations") is often taken to be a characteristic of Z itself, ignoring alternatives such as functional and algebraic styles. Z (ZRM or ISO-Z) provides a core language. Additionally, it is usual to use the Z Mathematical Toolkit, which adds many practical constructs to the core notation. This toolkit is generally assumed to be part of the core, and its scope mistakenly considered to impose fundamental restrictions (such as its definition of only *finite* sequences). It is common for Z specifications to be coerced into these conventions, no matter how inappropriate. By separating out and naming the Delta/Xi pattern and its associated subpatterns, and describing toolkit patterns, we hope to make it clear that these are just one choice of many.

A secondary motivation of our patterns work is to make more of the styles and levels accessible to developers so that appropriate choices can be made for each

application. For example, we have identified other architecture patterns that may be used instead of Delta/Xi.

To reiterate, the primary purpose of our patterns is to assist in the purpose of communication. Later, we consider conversion of a Z description to match a pattern, and conversion between patterns with different aims. This would allow, for example, the recasting of a specification written in a style that enhances proof, into a style that is suitable for presentation to clients or software designers, or *vice versa*.

In common with other languages that are the subject of software engineering patterns, Z is an expressive notation that can be made accessible (for writing and reading) to novices, but is generally the province of experts. We too provide a catalogue and a standard pattern format incorporating a name and an intention.

We are only beginning to understand the power of patterns in Z; our catalogue headings and pattern formats are still developing.

3.2 Structure of our Z pattern descriptions

Each reference work has its own structure for describing patterns. We use the following structure.

- Name : Conveys the essence, and expands the community "vocabulary"
- **Intent** : A summary of what the pattern provides
- **Problem** : A detailed description of the problem in context
- **Example** : A specific instance of the problem
- **Solution** : A description of the structure that can solve the problem
- **Illustration** : An illustration of the effect of applying the solution
- **Constraint** : Something that affects the use of the pattern
- **Variants** : Modifications of the pattern for certain circumstances, particularly where ZRM and ISO-Z solutions differ
- **Related patterns** : Other patterns to be used with, or in place of, this one
- **Specimens** : References to the literature where the pattern is used (often only implicitly)
- ■ : Indicates the end of the pattern description

Because of Z's generality and power, there are often several ways to solve a problem, with some solutions being better in some contexts. We include *choice patterns*, describing these various solutions and when they are most appropriate.

Some patterns can be *elaborated* in more significant ways than are covered by the **Variants** heading, the elaborations being almost further patterns in their own right. We describe such elaborations in abbreviated pattern form after the main pattern.

3.3 Pattern naming conventions

Patterns are usually named using verb phrases. This supports the descriptive purpose of enriching one's design vocabulary with pattern names.

- what do I want to do? – "I want to comment the intent"
- what did I do? – "I diagrammed the structure"
- what should I do now? – "I should apply syntax and type checks"

Certain architectural patterns are well-known Z concepts that would be made more obscure by changing their names, for example, promotion, Delta/Xi. Also, domain patterns typically already have a well-known name in the form of a noun phrase.

Antipatterns are named with noun phrases (for example, *overlong name*) that express the pitfall.

Choice patterns are named with present participles, for example, modelling product types, to stress the active choice to be made when applying the pattern. The name of a choice pattern is followed by (choice).

The names of pattern elaborations, and many components of generative patterns, are noun phrases.

3.4 Visualisation patterns

There are many diagramming styles appropriate for summarising the structure of Z components in different styles. For example, Venn diagrams can be used

to represent almost any set-theoretic statement; state machines summarise event-based structures, data-flow diagrams can represent functional styles etc.

Visualisation languages are easier to define in the context of patterns, as the full generality of the specification language is restricted to certain usage styles. The specific diagrammatic notation can itself be thought of as a sub-pattern, supporting a particular architecture pattern. We have devised diagrammatic sub-patterns for the Delta/Xi and morph architecture patterns. The diagrammatic morph sub-pattern is described along with the main morph pattern, later. The diagrammatic Delta/Xi sub-pattern is described here, because we use it in the following chapter describing promotion. It is adapted from a notation proposed by [d'Inverno & Luck 2001].

Delta/Xi : diagram the structure

Intent: Summarise the structure of a Delta/Xi specification using a diagram.

Problem: Since a Z specification is presented 'bottom-up' (declaration before use) and can be factored into many pieces, it may become difficult to 'see the wood for the trees'.

Solution: Construct a diagram to record the structure of the state and operation schemas, highlighting any Delta/Xi-related patterns used.

Do not worry over-much about being consistent and complete, and about distinguishing every small difference: the purpose of the diagram is to give a graphical overview of structure, not to be an alternative formal notation.

The following components are recommended.

- distinguish schemas by purpose
  - draw state schemas as named rectangles
  - draw operation schemas as named hexagons
  - draw other data types as named parallelograms
  - schemas not defined in the specification may be used in the diagram, for clarity. Indicate these by a dashed box. (Use of the Delta/Xi : strict convention sub-pattern means that ΔS and ΞS boxes are always dashed. The occurrence of other dashed boxes might indicate a refactoring opportunity.)
- for schema inclusion, use solid arrows pointing from the including schema to

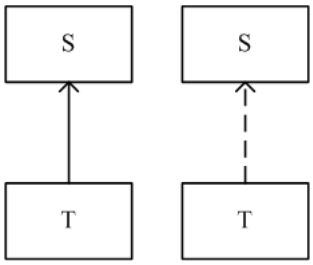

Figure 3.1 (a) schema T includes schema S. (b) schema T references schema S.

the included schema
  - for state inclusion, use a single line
  - for an operation including a state schema, S, via ΔS (and thus introducing a before- and an after-state), use a double line and a triangular 'Delta' arrowhead, pointing to the rectangle, S.
  - for an (initialisation) operation that includes only an after-state (S'), use a single line and an after-state $'$ by the arrowhead
  - an arrow directly to a box may be elided if there is an alternative path to that box
- indicate other uses of schemas by dashed lines from the referring data type to the referenced schema
- use highlighting (line thickness, box shading) to distinguish important parts of the diagram
  - if a description uses a pattern described with a diagrammatic form, the diagram of the description can be constructed by instantiating the structure of the pattern. Use highlighting to distinguish the pattern from other structural elements
  - highlight the full operations, as contrasted to intermediate definitions

As much as possible, without distorting the diagram, inclusion arrows are drawn upwards, so that the simplest schemas are at the top of the diagram, and constructs that are more complex are further down the page.

- If schema T includes schema S, either as declaration as $T == [\, S;\ \ldots \mid \ldots\,]$, or as a predicate as $T == [\ \ldots \mid S \land \ldots\ ]$, it can be drawn as figure 3.1a.
- If schema T refers to schema S other than by inclusion, for example as $T == [\, f : x \pfun S \ldots \mid \ldots\,]$, it can be drawn as figure 3.1b.

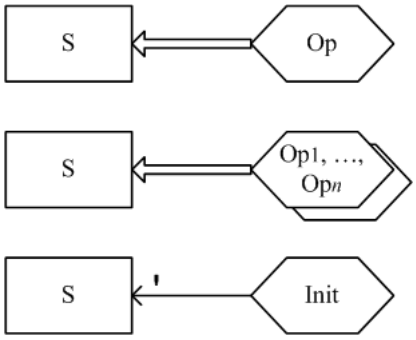

Figure 3.2 (a) operation *Op* includes ΔS. (b) several operations Op_i include ΔS. (c) initialisation operation *Init* includes S'.

- If operation schema *Op* includes schema S as $Op == [\ \Delta S \ldots \mid \ldots\ ]$, it can be drawn as figure 3.2a
- If multiple schemas S_i have precisely the same relationships with other schemas, their names can be listed in the same box, thereby drawing attention to their similar structures, as in figure 3.2b
- If initialisation schema *Init* includes schema S as $Init == [\ S' \ldots \mid \ldots\ ]$, it can be drawn as figure 3.2c

Illustration: Figures 4.1, 4.2 show the diagrams used to explain the promotion pattern, and figure 4.3 shows one of its variants; figures 15.1, 15.2, 15.3 show it applied to the refactored promotion example. Figures 11.1, 11.2 show the diagrams used to explain the refinement pattern. Appendix A.1 shows the diagrams of a large specification: the Purse specification structure.

Variants:

- The notation can be extended to show conjectures (as in the refinement pattern).
  - conjectures are drawn in an oval, labelled with a suitable name, pointing to any referenced schemas (see the example in Appendix A.1).
- For large specifications, a diagram may be split into sub-diagrams for clarity. It may be appropriate to draw a separate diagram for each operation, or family of operations, reproducing (relevant parts of) the diagram.
  - schemas or other data type boxes occurring in more than one sub-diagram are represented as rounded boxes except on the first occurrence (see the

example in Appendix A.1).

- If your application uses schemas in some particular way, extend the notation to capture your structure.

■

3.5 Catalogue of Z patterns

In using Z patterns, we have found many variations that were not immediately obvious. This could give rise to a very complex catalogue, and further work is needed on the existing categories. In writing about patterns, we use the following categories:

- Presentation patterns : ways of presenting, formatting and laying out Z specifications and documents
- Idiom patterns : styles of writing individual Z phrases
- Structure patterns : ways of structuring small pieces of Z specifications
- Architecture patterns : ways of structuring an entire specification
- Domain patterns : support for specific application domains
- Development patterns : assistance in parts of an engineering process, ranging from assistance in selecting appropriate formal methods and for applying formality at appropriate levels of rigour, to notation-specific development patterns for a particular system.

Under each category we also list certain *antipatterns*, which illustrate commonly occurring counter-examples to good style and best practice.

Themes re-occur in the different categories, and to some extent the divisions among categories are arbitrary. For example, patterns relating to naming and formatting exist at most levels. Patterns are context-dependent. So, for example, the particular details of a presentation pattern may be affected by the architecture, style and purpose of the Z description, and by the application domain.

CATEGORY	SUB-CATEGORY		
	Documentation	**Style**	**Usage**
Presentation	Comment the intention Provide navigation Name consistently *Overlong name* *Overmeaningful name*	Format to expose structure	
Idiom		Assemble from chunks Representing 1:many mappings Use free types for unions Making a schema binding Making a local declaration *Belated constraint* *Abused mu* *Bemused lambda* *Overloaded numbers*	
Structure	Name meaningful chunks Name predicates	Use generics to control detail *Fortran* *Sørensen shorty*	Modelling optional elements Modelling product types Modelling membership or flags *Boolean flag* *Partial precondition*
Architecture		Morph Event traces Object Orientation Algebraic style Goldilocks chunks *Unsuitable Delta/Xi pattern*	`Promotion` `Delta/Xi`
Domain		Application oriented theory	Domain specific toolkits Schema operator toolkit
Development		Focus the formality Use integrated methods	`Do a refinement` Animate Do sanity checks Express implicit properties Apply syntax and type checking Prove rigorously Prove formally

Table 3.1 Z Pattern Catalogue. In this table we use particular fonts to indicate the patterns, the *antipatterns*, and those with associated `generative` patterns and elaborations.

Table 3.1 summarises the categorisation of patterns that we have identified so far.

The next chapter discusses the promotion pattern in detail. Later chapters summarise (in considerably less and variable detail) other patterns in the patterns catalogue. As this document develops over time, these patterns will be fleshed out in more detail, and new patterns added as they are uncovered.

3.6 Patterns for changing the purpose of a Z specification

Several patterns do not neatly fit into table 3.1. These are patterns that allow the purpose of a specification to be changed. We focus on patterns that make Z readable. However, readable Z may hamper proof, and is certainly not optimal for implementation. In general, readability requires redundancy. High-level conversion patterns are needed.

Convert to more provable Z

Intent: Make it easier to perform required proofs on a readable Z schema containing redundancy.

Problem: Readability patterns collect all the details of a concept in one place, but putting all the predicates relevant to a particular specification concept inside a state schema complicates proof by adding subgoals to every proof on that state. Some of the predicates are there only to improve readability.

Solution: Move redundant predicates into conjectures that need only be proved once.

Related patterns: A suitably well-developed application-oriented theory can also ease the proof task.

■

Convert to implementable Z

Intent: Make it easier to implement a system that meets the proof-friendly or readable Z pattern.

Problem: Readability requires redundancy; provability moves redundant predicates to conjectures; redundant components should not be implemented, and conjectures cannot be directly implemented.

Solution: Remove redundant predicates and conjectures

■

These patterns (and their inverses) are the bases for generative sets of patterns for refactoring specifications. The refactoring must give an equivalent specification.

For example, a schema that is readable and is refactored to aid proof must be semantically identical; only refactorings based on choice patterns are permitted. However, these strict refactorings could take place in a wider context of refactoring. For example, the removal of redundant predicates/conjectures could occur before or after refinement or refactoring to generic schemas. (Meaning preservation is considered further below.)

Chapter 4

The Promotion Pattern, and its sub-patterns

4.1 Introduction

Before launching into the full Z Pattern Catalogue, in this chapter we describe one particular pattern, promotion, in detail, showing the pattern, its generative subpatterns, and some of its variations. In the following chapter, we show how to use the generative subpatterns to generate a specification that exhibits the promotion pattern.

Promotion is an *elaboration* of the architectural Delta/Xi pattern. Given it is so common, we choose to class it as a pattern in its own right, too.

4.2 The Promotion pattern

Promotion

Intent: Specify a global system in terms of multiple instances of a local state, and of operations that manipulate a local state.

Problem: A system that is essentially a hierarchy of components has operations and state at different levels in the hierarchy. This is difficult to specify directly, requiring Z structures such as μ and θ schema bindings.

Example: Some of the examples are presented below, including a banking system comprising a collection of accounts; a system for managing a collection of electronic (smart card) purses; an operating system for managing processes on a smart card.

Solution: Build a specification for local state and operations and the global state as a composite of local states. Use framing schemas to promote the local operations, and use schema calculus to construct appropriate global operations.

Specimens: Unix File System [Morgan & Suffrin 1992] (first published example); [Woodcock & Davies 1996] (useful refinement laws).

■

Illustrating the promotion pattern reveals the four sub-patterns that form promotion's set of generative patterns. These are summarised using the **intent** and the **solution** only.

The illustration is documented according to the presentation pattern, comment the intent, and illustrated by diagramming the structure.

Promotion : local state and operations

Intent: Describe the state and operations for one instance of a multi-instance system.

Solution:

$$Local == [\ \text{state components} \mid \text{constraints}\ ]$$
$$LocalOp == [\ \Delta Local;\ \text{inputs and outputs} \mid \text{constraints}\ ]$$

■

Promotion : global state

Intent: Describe the global context of a multi-instance system.

Solution:

- the global state requires a set, ID of identities.

  $[ID]$

- gbl maps identities to instances of the local state.

  $Global == [\ gbl : ID \pfun Local\ ]$

■

Promotion : framing schemas

Intent: Provide a context that describes how the global state is updated by the results of local operations.

Solution: The context, or "frame" is written as a "framing schema". This establishes the relationship of the state of a local instance to the global state, then shows how the after-state of an operation on that local instance is used to update the global state (the after-state of any suitable global operation) – no details of the global operation or the local operation are required; the frame merely deals with states established by operations defined elsewhere.

The following frame is for any operation that updates the state(s).

- $\Delta Global$ and $\Delta Local$ introduce the local and global states, pre- and post-operation.
- $x?$ is the global identity of an (existing) local instance.

$\Phi Update$

$$\Delta Global$$
$$\Delta Local$$
$$x? : ID$$

$$x? \in \mathrm{dom}\, gbl$$
$$\theta Local = gbl\ x?$$
$$gbl' = gbl \oplus \{x? \mapsto \theta Local\ '\}$$

- $x?$ is ascertained to be in the global set of local instances.
- The before-state of *Local* is the current local state associated with the identity, $x?$.
- The global state is updated by overriding the *gbl* function with the maplet from $x?$ to the after-state of *Local*.

The frame for introducing a new local instance to the global state is similar to an initialisation.

- Declarations are as before, except that there is only an after-state for *Local*.

ΦNew

$\Delta Global$
$Local'$
$x? : ID$

$x? \notin \operatorname{dom} gbl$
$gbl' = gbl \cup \{x? \mapsto \theta Local'\}$

- $x?$ does not have a mapping in the global state (it is an unused identity).
- The *gbl* function is updated by set union, adding the mapping from $x?$ to the after-state of *Local*.

The frame for an operation to remove a local instance from the global state is similar, but removes the local instance from the global mapping. The frame is only needed where the removal of a local instance must be verified using operations at the local level (see [Barden *et al.* 1994, chapter 19] for details).

■

Promotion : global operations

Intent: Write a global operation in terms of the defined local operations.

Solution: Use schema calculus to extract the relevant local operation (or after-state of a local operation) and conjoin the appropriate framing schema.

$$GlobalOp == \exists \Delta Local \bullet \Phi Update \land LocalOp$$
$$GlobalNew == \exists Local' \bullet \Phi New \land InitLocal$$

Again, deletion is composed in a similar way.

■

Promotion : diagram the structure

Intent: Summarise the structure of the promotion pattern using a diagram.

Solution: The diagram of the structure of promotion is shown in figures 4.1 and 4.2. Figure 4.1 shows the structure for the update operations; figure 4.2 shows the special form for the initialisation (creation of new local instances).

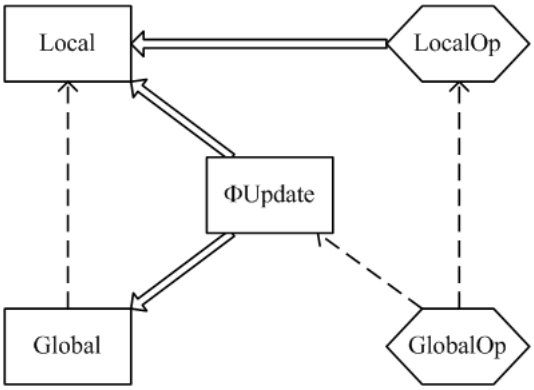

Figure 4.1 Structure of the promotion pattern.

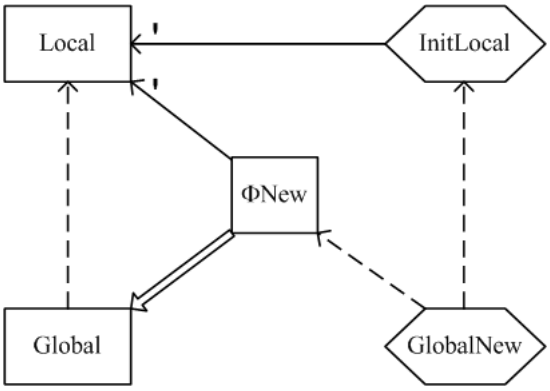

Figure 4.2 Structure of the promotion pattern, for the initialisation operation.

■

4.3 Elaboration Patterns for Promotion

The promotion pattern has several well-known elaborations, for coping with cases that do not fit the plain pattern.

Promotion (elaboration): global constraints

Intent: Add global state components to the collection of local instances

Solution:

GlobalwithState

Global
purely global state components

---

constraints on global state
constraints relating local states

The addition of global state components may require addition of constraints (a) on the global state, and (b) relating the local instance states; the promoted local operations may affect the global state through such constraints. There may also be operations that act on the global state components alone.

Example: [Stepney *et al.* 2000] defines a collection of concrete (ie refined from the abstract) electronic purses. The global state has additional mechanisms to control interoperability of the purses. The extended global state is:

- *conPurse* is the global state for the local purses, *ConPurse*, using identities from the set, *NAME*.
- *ether* and *archive* are global-only state elements.

ConWorld

$conPurse : NAME \twoheadrightarrow ConPurse$
$ether : \mathbb{P}\, Message$
$archive : NAME \leftrightarrow Message$

$\ldots$
$\mathrm{dom}\, archive \subseteq \mathrm{dom}\, conPurse$

- the global state element, *archive*, is constrained to only known local purses.

All the single purse operations are promoted to the global world level. However, the inputs and outputs from the local operation are not simply mapped into global inputs and outputs; they are also linked to the *ether* (the message transport mechanism) by the framing schema, which extends the framing schemas pattern:

$$
\begin{array}{|l}
\Phi UpdateCon \\
\Delta ConWorld \\
\Delta ConPurse \\
m?, m! : MESSAGE \\
name? : NAME \\
\hline
name? \in \mathrm{dom}\, conPurse \\
m? \in ether \\
\theta ConPurse = conPurse\ name? \\
conPurse' = conPurse \oplus \{name? \mapsto \theta ConPurse\ '\} \\
ether' = ether \cup \{m!\} \\
archive' = archive \\
\end{array}
$$

In addition to the promoted operations, there are some global-only operations, for example to add messages in the *ether* to the *archive*.

■

Promotion (elaboration): internal identifiers

Intent: Use a native element of the local state as the identifier.

The global identifiers used in promotion are arbitrary. Where the local state has its own identity, this may be a suitable global identifier.

Solutions: There are two solutions, depending on whether the promotion uses only internal identity, or uses both internal and global identity redundantly.

Solution 1: Internal identity only

Use Promotion : local state and operations thus:

- identity is included as a native attribute of the local state.

  $Local == [\ self : ID;\ \ldots\ ]$

- local operations must include a constraint that the identity is not changed by the operation.

  $LocalOp == [\ \Delta Local;\ \ldots \mid self' = self \land \ldots\ ]$

Use Promotion : global state thus:

- there is no global identifier; the global state is the set of local instances.

$$
\begin{array}{|l}
\hline
\textit{Global} \\
gbl : \mathbb{P}\, Local \\
\hline
\forall\, x, y : glb \mid x \neq y \bullet x.self \neq y.self \\
\hline
\end{array}
$$

- different local states have different identities

Global operations must include constraints to check the identity of local instances. For instance, the frame for the update operations modifies the predicates of the framing schemas pattern as follows.

$$
\begin{array}{|l}
\hline
\Phi\, Update \\
\Delta\, Global \\
\Delta\, Local \\
x? : ID \\
\hline
\exists\, x : gbl \bullet x.self = x? \\
\theta Local = (\ \mu\, Local \mid \theta Local \in gbl \land self = x?\ ) \\
gbl' = (gbl \setminus \theta Local) \cup \theta Local\,' \\
\hline
\end{array}
$$

The frame for the operation to add a new piece of local state is as follows.

- the existence of the instance has to be checked by seeking an element of *gbl* that has the given identifier, $x?$
- the local instance uses μ to bind to the relevant element of *gbl*
- the update replaces the local instance in the global set of instances with the result of the local operation

$$
\begin{array}{|l}
\hline
\Phi\, New \\
\Delta\, Global \\
Local' \\
x? : ID \\
\hline
\forall\, x : gbl \bullet x.self \neq x? \\
self' = x? \\
gbl' = gbl \cup \{\theta Local\,'\} \\
\hline
\end{array}
$$

- no pre-existing local state has the given identifier, x?
- the new local state has identifier x?
- the update adds the new local state to the global set

Solution 2: External and internal identities

The local definitions are the same as for the internal-only identifiers.

Use Promotion : global state thus:

- the global function is an injection, because there is a one-to-one correspondence between 'external' and 'internal' names

$$
\begin{array}{|l}
\textit{Global} \\
\hline
gbl : ID \rightarrowtail Local \\
\hline
\forall\, x : \operatorname{dom} gbl \bullet (gbl\ x).self = x \\
\hline
\end{array}
$$

- for any instance, external and internal identities are the same.

Illustration: In the concrete specification of the electronic purse, the purse has a name; this is the name that the purse is known by at the external level. The local state is *ConPurse*:

$$ConPurse == [\ name : NAME;\ \ldots \mid \ldots\ ]$$

Use Promotion : global state thus:

- *ConWorld*, uses the same type as the local identifier for the domain of the global *conPurse*.

$$
\begin{array}{|l}
\textit{ConWorld} \\
\hline
conPurse : NAME \twoheadrightarrow ConPurse \\
\ldots \\
\hline
\forall\, n : \operatorname{dom} conPurse \bullet (conPurse\ n).name = n \\
\ldots \\
\hline
\end{array}
$$

- every purse's internal name must be the same as the name by which it is known in the global system.

The framing schemas pattern is applicable, as the additional constraint is carried forward in the $\Delta ConWorld$ inclusion.

Specimen : Hall's style [Stepney *et al.* 1992, chapter 3] combines both these variants. It has a set of local states, and it has a *derived* mapping from (external) identities to local states:

HallStyle

$$gbl : \mathbb{F}\, Local$$
$$idGbl : ID \pfun Local$$
$$idGbl = \{\ l : gbl \bullet l.self \mapsto l\ \}$$

■

Promotion (elaboration): combine promotions

Intent: Specify a system that conforms to the local-global format for the promotion pattern, but has different sorts of local instance.

Solution:

- Each sort of local instance is associated with a separate component of the global state.

Global

$$gbl1 : ID1 \pfun Local1$$
$$\ldots$$
$$gbln : IDN \pfun LocalN$$

- Constraints could be added as appropriate.

Illustration:

The specification of a Smart Card Operating System [Stepney & Cooper 2000] specifies three types of application instances that the operating system has to control: fixed ISO applications, user programmable applications, and (for modelling reasons) 'absent' applications. For technical reasons to do with a particular proof,

in the specification these three types were combined using a free type, then promoted in the simple manner. However, it would be possible to promote the three types of local instances individually:

- *fixed* and *user* follow the global state pattern.
- *absent* has no local state other than an identifier, so the internal identifier pattern of promotion is applied.

$$
\begin{array}{|l}
\hline
\textit{CardGlobal} \\
\textit{fixed} : \textit{ID} \pfun \textit{Fixed} \\
\textit{user} : \textit{ID} \pfun \textit{Appl} \\
\textit{absent} : \mathbb{P}\, \textit{ID} \\
\hline
\mathrm{disjoint}\langle \mathrm{dom}\, \textit{fixed}, \mathrm{dom}\, \textit{user}, \textit{absent} \rangle \\
\hline
\end{array}
$$

- The global identifiers for each promotion are drawn from the same set, *ID*, so are constrained to be disjoint.

■

Promotion (elaboration): multi-promotion

Intent: Specify a system that comprises multiple instances, but has global operations that may affect more than one local instance.

Solutions: The local state and operations and global state patterns are applied. The framing schemas include multiple local instances. There are two possible patterns. The first applies when the global operations affect a fixed number of local instances. The second is more general, and applies when the number of local operations affected by the global operation is variable or unknown.

Solution 1: A fixed number of local instances (illustrated for two instances, but extensible to as many as are practical or desired).

Use Promotion : framing schemas thus:

- The (otherwise identical) local states are declared twice, distinguished by decorations. (Note that in $\Delta \textit{Local}_1$, the decoration applies to the whole $\Delta \textit{Local}$ phrase, and thus yields $\textit{Local}_1$ and $\textit{Local}'_1$.)

$$
\begin{array}{|l}
\hline
\Phi Update \\
\Delta Global \\
\Delta Local\ _1 \\
\Delta Local\ _2 \\
x?, y? : ID \\
\hline
\{x?, y?\} \subseteq \mathrm{dom}\, gbl \\
x? \neq y? \\
\theta Local\ _1 = gbl\ x? \\
\theta Local\ _2 = gbl\ y? \\
gbl' = gbl \oplus \{x? \mapsto \theta Local\ '_1, y? \mapsto \theta Local\ '\ _2\} \\
\hline
\end{array}
$$

- the provided identifiers are different instances from the set of identifiers used in the global system
- each local state instance is bound to one of the input identifiers
- the result of the global operation overrides the global state with the mappings from each identifier to the after-state of the local operation on its local state

Then the global operation becomes

$$
GlobalOp == \exists \Delta Local\ _1;\ \Delta Local\ _2 \bullet \\
\qquad \Phi Update2 \land ALocalOp\ _1 \land AnotherLocalOp\ _2
$$

Illustration 1: This pattern is illustrated in the specification of the electronic purse transfer operations, chapter 15.

Figure 4.3 gives a diagram of the underlying structure of a multi-promotion, based on the single promotion diagram. The relevant inclusion arrows are annotated with the number of local states and operations.

Solution 2: An arbitrary number of local instances

If there is an arbitrary number of local states all affected by the same operation, they can be bundled into a sequence, and the framing schema becomes

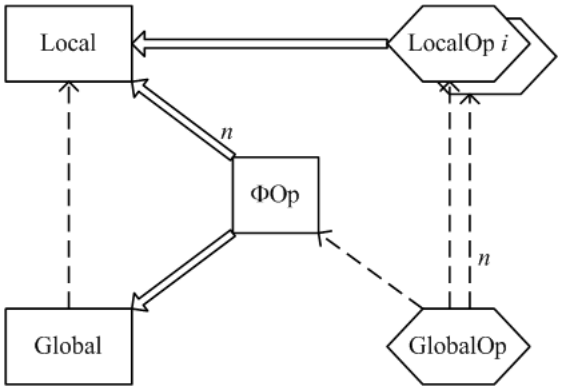

Figure 4.3 Structure of the promotion pattern, with the multi-promotion elaboration, fixed number of local states.

$$
\begin{array}{|l}
\hline
\Phi\mathit{UpdateAll} \\
\Delta\mathit{Global} \\
\mathit{ls} : \operatorname{seq} \Delta\mathit{Local} \\
\mathit{xs}? : \operatorname{iseq} \mathit{ID} \\
\hline
\#\mathit{xs}? = \#\mathit{ls} \\
\operatorname{ran} \mathit{xs}? \subseteq \operatorname{dom} \mathit{gbl} \\
\forall\, i : \operatorname{dom} \mathit{ls} \bullet \exists\, \Delta\mathit{Local} \mid \theta\Delta\mathit{Local} = \mathit{ls}\ i \bullet \theta\mathit{Local} = \mathit{gbl}(\mathit{xs}?\ i) \\
\mathit{gbl}' = \mathit{gbl} \oplus \{\ i : \operatorname{dom} \mathit{ls};\ \Delta\mathit{Local} \mid \theta\Delta\mathit{Local} = \mathit{ls}\ i \bullet \mathit{xs}?\ i \mapsto \theta\mathit{Local}\ '\ \} \\
\hline
\end{array}
$$

The local operation is cast into a form for use with the framing schema (assuming it has inputs and outputs declared as $\mathit{in}? : \mathit{IN}, \mathit{out}! : \mathit{OUT}$):

$$
\begin{array}{|l}
\hline
\mathit{LocalOpS} \\
\mathit{ls} : \operatorname{seq} \Delta\mathit{Local} \\
\mathit{ins}? : \operatorname{seq} \mathit{IN} \\
\mathit{outs}! : \operatorname{seq} \mathit{OUT} \\
\hline
\#\mathit{ls} = \#\mathit{ins}? = \#\mathit{outs}! \\
\forall\, i : \operatorname{dom} \mathit{ls} \bullet \\
\quad \exists\, \Delta\mathit{Local};\ \mathit{in}? : \mathit{IN}, \mathit{out}! : \mathit{OUT} \mid \\
\qquad \theta\Delta\mathit{Local} = \mathit{ls}\ i \land \mathit{in}? = \mathit{ins}?\ i \land \mathit{out}! = \mathit{outs}!\ i \\
\qquad\quad \bullet\ \mathit{LocalOp} \\
\hline
\end{array}
$$

Then the global operation becomes

$$GlobalOp == \exists\, ls : \operatorname{seq} \Delta Local \bullet \Phi UpdateAll \land LocalOpS$$

Illustration 2: Consider a local state, *Counter* that holds an incrementable natural number. An operation on the local state resets the number to zero. Following the local state and operations pattern, these are defined:

$$Counter == [\ c : \mathbb{N}\ ]$$
$$Reset == [\ \Delta Counter \mid c' = 0\ ]$$

A global system of counters can be defined using the global state pattern:

$$CounterSystem == [\ sys : NAME \pfun Counter\ ]$$

To simultaneously reset all the identified counters to zero (assuming a $\Phi UpdateAll$ defined following the pattern above, but using the names of this example):

ResetS

$$ls : \operatorname{seq} \Delta Counter$$

$$\forall\, i : \operatorname{dom} ls \bullet \exists\, \Delta Counter \mid \theta \Delta Counter = ls\ i \bullet Reset$$

$$ResetAll == \exists\, ls : \operatorname{seq} \Delta Counter \bullet \Phi UpdateAll \land ResetS$$

This is equivalent to

ResetAll

$$\Delta CounterSystem$$
$$xs? : \operatorname{iseq} NAME$$

$$\operatorname{ran} xs? \subseteq \operatorname{dom} sys$$
$$\forall\, x : \operatorname{ran} xs? \bullet$$
$$\quad \exists\, \Delta Counter \bullet$$
$$\quad\quad \theta Counter = sys\ x \land Reset \land sys'\ x = \theta Counter\ '$$

■

Chapter 5

Generating a Promotion, by example

5.1 Introduction

A generative pattern is one that helps us to *construct* a specification, rather than simply describing a structure that a specification could have. We describe the generative promotion pattern by the example of a bank account system.

5.2 Starting point: a local state

The starting point for applying the generative pattern is an existing specification of the local state, here a single bank account:

- A *limit* is defined as an integer. It represents a global constant for the specification.

  $$limit : \mathbb{Z}$$

- A bank account has an integer *balance*.

  Account

  $$balance : \mathbb{Z}$$
  $$\ldots$$
  $$limit \leq balance$$

- At any time, the account balance must be greater than the (lower) limit for that account.

Various operations are defined on this single account:

$$AccountOp == [\ \Delta Account;\ \ldots \mid \ldots\ ]$$
$$InitAccount == [\ Account\ ';\ \ldots \mid \ldots\ ]$$

The aim is to specify the whole bank account system, comprising many such single accounts. Clearly, this matches the intent of the architecture pattern for promotion:

> "Specify a global system in terms of a collection of local states, and of operations that manipulate the local states."

To construct the bank specification, we therefore apply the generative patterns for promotion. We have a local state, but it does not exactly fit the local state and operations pattern, because it requires the global constant, *limit*. Refactoring is used to preserve the meaning of the existing specification, but to make the specification match the generative pattern.

5.3 Step 1: refactor global attributes into the local state

In the account specification, the *limit* is intended to be different for different accounts. As part of the first refactoring step, it is moved into the local state[1].

Account

$$balance : \mathbb{Z}$$
$$limit : \mathbb{Z}$$
$$\ldots$$
$$limit \leq balance$$

The refactoring must preserve the meaning of operations as well as of the state. In the original specification, *limit* cannot be changed because it is declared axiomatically. So in the refactored specification, every operation requires a predicate stating that the limit does not change. This could use the Delta/Xi : change part

[1] Note that, if the bank set one limit that was applicable to every account, this could be defined as part of the global state.

of the state pattern, but as there is only one simple variable involved, the predicate is included explicitly.

$$AccountOp == [\ \Delta Account;\ \ldots \mid limit = limit' \land \ldots\ ]$$

The specification now matches the local state and operations pattern.

5.4 Step 2: introduce global state

Following the set of generative patterns, the global state is defined as a collection of local states. No identifiers are included in the simple account definition, so a given set is used at the global level. The global state representing the bank's accounts is thus a mechanical application of the global state pattern:

$$[ID]$$
$$Bank == [\ acc : ID \pfun Account\ ]$$

5.5 Step 3: define framing schemas

Reference to the generative patterns shows that a separate framing schema is needed for operations that create, remove and change local instances of account. These use the three versions of the framing schemas pattern. The frame for operations that change a local instance is given. (The other framing schema is for an account creation, $\Phi NewAccount$.)

$\Phi Update$

$\Delta Bank$
$\Delta Account$
$anId? : ID$

$anId? \in \mathrm{dom}\ acc$
$\theta Account = acc\ anId?$
$acc' = acc \oplus \{anId? \mapsto \theta Account\ '\}$

5.6 Step 4: define global operations

The global operations pattern is used to express the system operations:

$$BankOp == \exists \Delta Account \bullet \Phi Update \land AccountOp$$
$$NewAccount == \exists Account' \bullet \Phi NewAccount \land InitAccount$$

There is also an operation, *CloseAccount*. This is a global operation that removes an instance of account from the global state. It does not require a framing schema, so long as there are no local side conditions on the deletion:

$$CloseAccount == [\ \Delta Bank \ldots\ ]$$

Note that if there were local side conditions, such as the need to check that the balance was zero, a framing schema would be used.

This results in the promoted bank account specification.

■

5.7 Further promotions

Application of the generative patterns has produced, merely by elaboration of templates, a bank specification that is a promotion of the simple account specification. Note that the ending point of this generation is a specification matching the Delta/Xi pattern. This could itself become the local state for a promotion, if the next requirement were to specify a chain of banks. In this way, promotion can be used to generate any hierarchical structure of local states and operations.

Chapter 6

Presentation patterns

6.1 Introduction

Presentation patterns are analogous to low-level coding standards: how to comment, how to cross reference, how to format. These patterns seem self-evident to people used to structured programming or trained to follow company styles for presentation. However, much published Z is not presented in a consistent manner. The pattern solutions given here are based on experience of constructing and proving large formal texts, and the needs of the checkers and reviewers of these documents (eg [Stepney *et al.* 2000]).

Presentation patterns dictate presentational conventions for the Z documents and components. The pattern details given here relate to Z written for the purpose of communication. Whilst the patterns are still relevant to Z written for other purposes, the detailed recommendations would be different. Similarly, several patterns are illustrated for Z written according to the Delta/Xi pattern; the details of these patterns would also change if the style of Z were different.

6.2 The presentation patterns

Comment the intention

Intent: Communicate the intent of every part of the specification, with a uniform commenting style.

Problem: A Z specification that comprises only mathematics is not readable or maintainable. The mathematics provides a reasonably unambiguous specification,

but the variable names are an insufficient link to the real world.

Example : An unambiguous but obscure Z schema:

$$
\begin{array}{|l}
\hline
\textit{Memory} \\
\textit{ram}, \textit{rom} : \textit{ADDR} \pfun \textit{BYTE} \\
\textit{inmap}, \textit{outmap} : \mathbb{P}\, \textit{ADDR} \\
\hline
\mathrm{disjoint}\langle \mathrm{dom}\, \textit{ram}, \mathrm{dom}\, \textit{rom} \rangle \\
\textit{inmap} \cup \textit{outmap} \subseteq \mathrm{dom}\, \textit{ram} \\
\mathrm{disjoint}\langle \textit{inmap}, \textit{outmap} \rangle \\
\hline
\end{array}
$$

Solution: Provide a commentary in a particular style at a number of levels. The text needs to be written and maintained with as much care as the Z, and must add to the value of the mathematical statements: the text is not just a "translation" of the maths.

- If necessary, include an introductory overview of the domain, as context for the specification; use diagrams to capture the architecture.
- In the body of the specification, write a short sentence that conveys the intent of every Z paragraph, linking to the real world; if the Z is long, provide further commentary describing the intent of the internals. Do not simply "translate" the Z into natural language in the comment: if the Z says $x' = x + 1$, a comment like "increment x by 1" adds little value; a comment like "x counts the number of input events" is better.
- Where a Z paragraph has internal structure (for example, an axiomatic paragraph or schema has a declaration part and a predicate part; a free type definition has a number of branches), let the comment structure clearly follow that of the Z paragraph, with a readily identifiable part of the comment provided for each part of the paragraph. For a schema, for example, precede the Z paragraph with commentary on declarations, follow the paragraph with commentary on the internal predicates. Use bullet lists to help distinguish the separate parts.

Illustration:

The schema *Memory* defines the memory state of the device.

- *ram* describes the dynamic memory, as a mapping from memory addresses to the byte values they contain

- *rom* describes the read-only memory, as for *ram*
- *inmap*, *outmap* are the memory-mapped input/output memory locations

$$
\begin{array}{|l}
\textit{Memory} \\
\hline
ram, rom : ADDR \pfun BYTE \\
inmap, outmap : \mathbb{P}\, ADDR \\
\hline
\text{disjoint}\langle \text{dom}\, ram, \text{dom}\, rom \rangle \\
inmap \cup outmap \subseteq \text{dom}\, ram \\
\text{disjoint}\langle inmap, outmap \rangle \\
\hline
\end{array}
$$

- *ram* and *rom* have distinct address spaces
- the memory mapped i/o lies within *ram*
- no address is both input and output

■

Format to expose structure

Intent: Expose the structure of each Z phrase by consistent and clear positioning of its component parts.

Problem: Arbitrary formatting of complicated phrases makes a Z document hard to read, because it hinders the readers' "pattern matching" abilities for recognising and understanding structure.

Solution: Define and use a consistent formatting style, for example:

- break lines according to the syntactic structure of the phrase: break lines at nodes higher up the syntax tree before breaking at lower nodes
- break lines so that operators are at the beginnings of lines, where they are more visible
- indent to clarify the structure – increase the indentation of a broken line and align indentation of parts at similar levels
- place a consistent amount of white space around operators and other small constructs; place more around relational operators than functions and generics

- place more whitespace around the braces in set comprehensions and around the square brackets in horizontal schemas than around the corresponding brackets in set extensions and other bracketed constructs

Specimens: Example styles are used in [Stepney *et al.* 2000], and in the CADiℤ [Toyn 2001] and Formaliser [Stepney 2001] pretty-printers.

■

Provide navigation

Intent: Provide supplementary material to allow the reader to locate the declaration and uses of any name in the Z document.

Problem: Real specifications can be large (several hundred pages), and a global name may be used far from its declaration. That declaration provides context, type information, and constraints, all of which are needed to understand the name. Additionally, particularly when modifying a specification, an investigation of all uses of a name is important. Tools provide on-line navigation (from full hyperlinking, down to using `grep`), but it can be hard to discover the appropriate declaration or all uses in a large paper document.

Solution: Provide a complete index of global name declarations. Global names include schema components, indexed as, for example, *compName*; *SchemaName*. If a specification is split across several documents, perhaps with a domain-specific toolkit in a separate document, provide indexing into all the documents. (The well-known Mathematical Toolkit names do not usually need to be indexed.) Consider also providing an index of where global names are used.

Related patterns: Name consistently, to give clues to the meaning of a name. At the architectural levels, Name meaningful chunks and Use Goldilocks chunks ensure any local names have use and declaration simultaneously visible.

Specimens: [Stepney *et al.* 2000] has a full index; the CICS project [Wordsworth 1987] uses marginal cross-referencing. [Arthan 2000]'s convention is to use **bold** font when declaring a global name, then ordinary font when using it.

■

Name consistently

Intent: Use a naming convention that conveys structural information.

Problem: Names are important, and arbitrary component names can confuse and disorient the reader. Even an originally good naming convention can be eroded during maintenance.

Solution: Define, document and use a naming convention for a specification. The convention may include type information, scope information (for example, short names only for local scope or generic formal parameters), intent information (as given by comment the intent), and a link to other names with a related intent (for example, by consistent use of prefixes, suffixes or subscripts).

Related patterns: Delta/Xi has a convention for inputs, outputs and state changes. Promotion has a convention for framing schema names. A complex convention can lead to the *overlong name* antipattern.

Specimen: ZIP [Barden *et al.* 1994, chapter 8] introduces a simple convention for typography of names in a Z specification.

■

6.3 Presentation antipatterns

Overmeaningful name

Intent: Properties implied by names should not be relied upon until made explicit in the Z text.

Problem: An apparently meaningful name is declared, but is never constrained to have the property its name suggests.

Example: A declaration $even : \mathbb{N}$ is made, and is never followed by any further constraint on the values of $even$.

Solution: Do not build more into the name than the mathematics can deliver. The name should be 'meaningful' in the way it links to the real world, not 'over-meaningful' in implying restrictions that do not actually hold.

Variants: When using lightweight formality, it can be useful to build meaning into the name that is not fully supported by the intentionally-abbreviated mathematics.

In such cases, comment the intent carefully, pointing out this fact.

■

Overlong name

Intent: Use names that enhance the readability of the specification.

Problem: Long names can cause readability problems: they may cause overlength lines and linebreaks that detract from readability; if they differ by only a few letters, they may be difficult to distinguish; they may hinder the reader's recognition of the mathematical patterns in use.

Solution: Use names that are long enough to be meaningful, but short enough to be readable. Make similar names more distinguishable by placing their differences at the beginning, rather than the end, of the word.

■

Chapter 7

Idiom patterns

7.1 Introduction

Z is a powerful notation, and there can be several ways of achieving a particular end. It is often useful to choose a particular idiomatic way of doing something, and sticking to it. The idiom itself then becomes part of the vocabulary.

7.2 The idiom patterns

Assemble from chunks

Intent: Construct a complex Z structure from understandable chunks.

Problem: There can be a large gap between the high level concepts that we need to express and the low level language elements that Z provides for their expression. Often a complicated set needs to be defined, with only fairly primitive constructors available.

Solution: Apply the "divide and conquer" principle. Specify simple component sets that make sense in isolation, and then combine them using Mathematical Toolkit operators, domain-specific toolkit operators, or schema calculus, as appropriate, into the complicated whole.

This "assembly" might join parts together (union, schema disjunction, concatenation), remove parts (intersection, schema conjunction, set difference, restriction, projection, filtering), supersede some parts (relational override), connect parts stepwise (relational and schema composition, piping, relational closure), and so

on. Ensure that the simpler parts themselves make sense in isolation ("all small or medium widgets, except blue ones").

Constraint: Sometimes things really are so baroque that they cannot be broken down into simpler chunks any bigger than individual elements. If this is so, ask why. If there is no need to follow an existing implementation or standard, try simplifying the specification so that it can be chunked. At least try to isolate the baroque part in a chunk of its own, with copious commentary: *lotsOfEasyStuff* $\oplus$ *smallButHorrible*.

Related patterns: At higher structural levels, equivalent compositional patterns are name meaningful chunks and Goldilocks chunks. The naming of chunks uses the presentation pattern name consistently. The goal of the Z text is emphasised by use of the presentation pattern comment the intention, at both the component and composite levels, but paying particular attention to any *smallButHorrible* chunk. The Delta/Xi : disjoin errors sub-pattern uses the schema calculus to combine various sub-operation schemas.

■

Representing 1-many mappings (choice)

Intent: Choose how to model a relationship with a set of target values.

Problem: Each element of X is related to a set of values of Y. There are several ways to model this.

Solution choice:

1. Representing 1-many mappings : using a target set
   $f : X \pfun \mathbb{P}\, Y$: this precisely expresses the problem, and is most useful if the various ys associated with a particular x are being manipulated explicitly: they are accessible as $y \in f\ x$. This formulation is difficult to use in a more algebraic style, where there is frequent need to flatten sets of range sets using generalised unions.
2. Representing 1-many mappings : using a relation
   $fr : X \rel Y$: this expresses the underlying structure, and tends to be most useful if the relation is being manipulated as a whole, in an algebraic style. Extraction of explicit range elements requires use of the relational image, $fr(\!|\ \{x\}\ |\!)$, in place of simple function application, $f\ x$.

3. Representing 1-many mappings : using the inverse mapping
   $finv : Y \pfun X$: if the inverse is a function, this may be a better model of the relationship.

Constraint: The first two choices are not exactly isomorphic: the functional form can distinguish absence from the mapping, $x_1 \notin \operatorname{dom} f$, from mapping to the empty set, $f\ x_2 = \varnothing$, whereas the relational form cannot.

■

Making a schema binding (choice)

Intent: Specify a particular schema binding.

Problem: A particular schema binding has to be specified, that is, values need to be associated with the variables (*foo* and *bar*) of an instance of the schema (θS). There are several ways to do this; one involves using μ in an idiomatic way.

Solution choice:

1. Making a schema binding : ISO-Z
   In ISO-Z, schema bindings can be written directly, as in $\langlebar\ foo == x, bar == y \rangle\!\rangle$,
2. Making a schema binding : ZRM
   In ZRM, schema bindings cannot be easily written; a mu-expression is often used, as in $\mu\, S \bullet foo = x \land bar = y$.

■

Making a local declaration (choice)

Intent: Use local declarations in an expression or predicate.

Problem: There are several ways to do this; one involves using μ in an idiomatic way.

Solution choice:

1. Making a local declaration : let-expression
   In ZRM and ISO-Z, a let-expression can be used, as in $\operatorname{let} x == foo;\ y ==$

bar • *expr*. In ISO-Z a μ expression can be used in an idiomatic way to get an equivalent construct, as in $\mu\, x == foo;\ y == bar \bullet expr$.

2. Making a local declaration : ZRM let-predicate
   In ZRM, a let-predicate can be used, as in $\mathrm{let}\, x == foo;\ y == bar \bullet pred$.
3. Making a local declaration : ISO-Z quantifier
   ISO-Z has no let-predicates; an existential quantifier can be used in an idiomatic way to get an equivalent construct, as in $\exists\, x == foo;\ y == bar \bullet pred$.

■

Use free types for unions

Intent: Bundle together different types so they can be treated uniformly.

■

7.3 Idiom antipatterns

Belated constraint

Intent: Avoid separating named elements and parts of their definition.

Problem: A global name is declared and, much later in the specification, is constrained. Any use of the name between declaration and constraint is subject to this constraint, but the reader is not yet aware of it.

Example: A global name is declared, say as $n : \mathbb{N}$. n is used liberally in the specification, but something seems a little off key. We read on, hopefully, and many pages later, we discover the global constraint $n \in even$. All our laboriously built-up understanding of the specification is destroyed in that one line (or, more likely, it suddenly becomes clear why nothing really made sense until now!)

Solution: Specify any constraints along with the declaration, in the same paragraph or as close to it as possible. If it really is not possible to write the constraint with the declaration, put a clear comment at the declaration, with a forward reference to the constraint.

Constraint: Some proof tools require each paragraph to be a *conservative extension* of the specification, because that makes certain sanity check proofs easier; thus they disallow belated constraints. Also, purposes other than communication may dictate different patterns (see convert to provable Z, above).

■

Abused mu

Intent: Make the specification as accessible as possible to all levels of readers.

Problem: An expression containing μ is almost always unreadable, except to very experienced specifiers.

Solution: Confine the use of μ to idiomatic ways, such as making a schema binding (and possibly in making a local declaration, but let is to be preferred). If it is deemed necessary to use μ in other contexts, comment the intention very carefully.

Specimens: [Barden *et al.* 1994, chapter 24] describes some uses and abuses of μ.

■

Bemused lambda

Intent: Make the specification as accessible as possible to all levels of readers.

Problem: In some contexts, experienced formalists use unnamed lambda functions. This can considerably reduce the effort of specification, by making it clear that a function is being defined, and in not requiring the invention of a name. However, the *type* of the function is not explicit, and the use of lambda functions is often confusing to less experienced readers.

Solution: Comment the intention of such lambda functions very carefully. Consider whether to express implicit properties (of the function's type).

■

Overloaded numbers

Intent: Exploit the Z type system and typechecking tools to catch as many errors as possible.

Problem: Subsets of $\mathbb{N}$ or $\mathbb{Z}$ are often used as convenient models of labels and identifiers in specifications, but a typechecker cannot catch cases where the wrong kind of label is being used.

Solution: Use given sets and free types, rather than subsets of $\mathbb{N}$ or $\mathbb{Z}$, where possible.

Specimen: [Brown 1979] quotes his eighth deadly sin as "to use numbers for objects that are not numbers".

■

Chapter 8

Structural patterns

8.1 Introduction

At the intermediate level, structural patterns guide the selection and construction of components of the formal description. Whilst some of these patterns represent presentational issues, several capture solutions to potentially hard practical problems.

8.2 The structural patterns

Use generics to control detail

Intent: Use type representations that are consistent throughout a development but that, at any phase of development, carry the details relevant to only that level of abstraction.

Problem: Z texts need to refer to many kinds of real-world and/or implementation-level data. Full specification of the data types clutters high-level Z descriptions and usually over-constrains possible implementations. Introducing more concrete types in a lower-level specification forces an expensive data-refinement proof obligation. In some cases it is not possible to introduce the desired schema, because of a recursive structure.

Example: A specification needs to represent personal details and addresses. The representation depends on the level of detail required, and the planned use of the specification. At the abstract level, $[NAME, ADDRESS]$ is a sufficient model. At

a more concrete level, however, it becomes necessary to define a (possibly complex) internal structure for *NAME* and *ADDRESS*.

Solution: Make the specification generic in the types that will need to be elaborated. This abstract representation does not impose unnecessary constraints on the specification. Later, define concrete types with the additional detail, and use it to instantiate the generic specification.

Illustration: An abstract *Customer* has a *name* and *address*, of generic type.

Customer $[NAME, ADDRESS]$

$name : NAME$

$address : ADDRESS$

Some customers are personal (as opposed to companies) and have a *name* with the format of a *PersonalName* (as opposed to a company name, say), a particular concrete form.

PersonalName

$title, forename, surname : String$

Personal customers have a *HouseAddress*. Different countries use a different form of *CODE*, so that is left generic here.

HouseAddress $[CODE]$

$house : \mathbb{N}_1$

$street, town, country : String$

$code : CODE$

In the UK, the address code is a *Postcode* (details omitted)

$$Postcode == \ldots$$

So a *PersonalUKCustomer* has a *PersonalName*, and *HouseAddress* instantiated with a *Postcode*.

$$PersonalUKCustomer == Customer[PersonalName, HouseAddress[Postcode]]$$

Variants:

1. A generic schema can be used in a recursive free type definition (*suggested by* Rob Arthan).

$$Node[T] == [\ left, right : T;\ value : \mathbb{N}\ ]$$
$$TREE ::= leaf \mid node\langle\langle Node[TREE]\rangle\rangle$$
$$TreeNode == Node[TREE]$$

2. Type-constrained generics [Valentine *et al.* 2000] are an extension to Z that allows definitions to require their generic parameters to be instantiated only with schemas having certain signatures; this permits a wider use of generics in Z specifications.

Specimens: Instantiation of generics with more concrete types is demonstrated in [Polack & Stepney 1999]. Other variants appear in [Flinn & Sørensen 1992] and various toolkit definitions.

■

Modelling optional elements (choice)

Intent: Provide a type that allows variables to take an unknown or optional value.

Problem: Some system domains use a type-correct marker, a *null*, to represent a value that is known to be inapplicable in certain circumstances, but Z sets do not have such a concept, and using powersets with $null == \varnothing$ is over-complicated.

Example: A database relation comprises n-tuples, each of which has a value for each component attribute. If a particular attribute is not applicable to a particular n-tuple, the *null* value is inserted.

Solution choice:

1. Modelling optional elements : free type
   Use a free type to add the optional element

$$OPTTYPE ::= null \mid present\langle\langle TYPE\rangle\rangle$$

   Several different "null" elements can be accommodated, and can even have some structure, such as $null\langle\langle ID\rangle\rangle$. However, it clutters the name-space with new names for each new optional type, and has a wordy syntax.
2. Modelling optional elements : generic optional
   [d'Inverno & Priestly 1995] give a compact treatment of the problem. They

define a generic operator (on types) with the characteristics of the optionality; this can be instantiated in the declaration of any variable that can take an optional value.

$$optional[X] == \{\ a : \mathbb{P}\, X \mid \#a \leq 1\ \}$$

relation($defined$ _)
relation($undefined$ _)

$[X]$

$$defined\ \_, undefined\ \_ : \mathbb{P}\ optional[X]$$

$$\forall\, a : optional[X] \bullet$$
$$\quad defined\ a \Leftrightarrow a = 1$$
$$\quad \land\ undefined\ a \Leftrightarrow a = 0$$
$$\forall\, a : optional[X] \mid defined\ a \bullet$$
$$\quad the\ a = (\ \mu\, x : X \mid x \in a\ )$$

Being generic, this approach introduces the new names *optional*, *defined*, *undefined*, and *the*, to the name-space only once, not once for each new optional type.

■

Name meaningful chunks

Intent: Improve comprehensibility of a large or complex structure by building it from identifiable components.

Problem: A reluctance to separate out and name components that are used only once can lead to complex structures. The intent of the structure is not apparent without careful deconstruction.

Solution: Construct and name components which are meaningful in themselves, even if they are used only once.

Related Patterns: Assemble from chunks is the similar low level idiom pattern. Predicates are covered by name predicates. The Delta/Xi : disjoin errors sub-pattern uses the schema calculus to combine various named sub-operation schemas.

■

Name predicates

Intent: Name a predicate to help expose its meaning, and to allow its use by name elsewhere.

Problem: An interesting property might be used in a wider context, but it is difficult to understand in that context – it is getting swamped by other details. Also, the same property might be needed in several contexts.

Solution: Define the property as the predicate part of a schema, and use the reference to the schema wherever the property is needed. Use $[new/old]$ renaming if necessary.

Illustration:

$$Triangle == [\ a, b, c : \mathbb{N}\ ]$$
$$IsIsosceles == [\ Triangle \mid a = b \lor b = c \lor c = a\ ]$$
$$IsEquilateral == [\ Triangle \mid a = b = c\ ]$$
$$\vdash \forall\ Triangle \mid IsEquilateral \bullet IsIsosceles$$

Constraint: Take care that the declarations are of appropriate scope.

Related patterns: Name consistently: use the schema naming convention, with some additional convention to show it is a property. (Here, properties are named as IsX.) Also patterns assemble from chunks, name meaningful chunks.

Specimens: [Stepney *et al.* 2000] use the predicates *SufficientFundsProperty* and *Authentic* in the relevant operations to impose two required properties on the system.

■

Modelling product types (choice)

Intent: Choose the more appropriate product constructor for a context.

Problem: Z has two nearly-isomorphic product type constructors. Which one should be used in a given context?

Solution choice:

1. Modelling product types : schema
   Component names are meaningful (for example, *memory.register* would refer to the *register* component of the schema binding *memory*). This fits the intention of name meaningful chunks. However, constructing particular schema bindings is relatively verbose. Use a schema product where the names convey some helpful meaning, and where component selection is common.
2. Modelling product types : Cartesian product
   Components are located by position (for example, *memory*.3), which communicates little meaning. However, the tuple construction syntax is terse. Use a Cartesian product when terseness is valuable, for example, the main use of the product is writing explicit values, as in algebraic style operator definitions. Use a Cartesian product when there are no meaningful names to be had, for example, if the components are bundled into a tuple simply in orer to return multiple results from a function.

Specimens: of schemas: [Polack *et al.* 1993], [Mander & Polack 1995]; of tuples: many Mathematical Toolkit definitions

Variants: In ZRM, schema components can be directly selected, as in $S.foo$, but schema bindings cannot be directly written (a mu-expression is often used, following making a schema binding 2); tuples can be directly written, as in (x, y, z), but their components cannot be directly selected (a lambda-expression with a characteristic tuple is often used, as in $\lambda\, x : X;\ y : Y;\ z : Z \bullet z$). ISO-Z allows schema bindings to be directly written (see making a schema binding 1), and tuple components to be directly selected, as in $t.3$, but there is still a difference in the terseness of construction.

■

Modelling membership or flags (choice)

Intent: Provide a meaningful distinction between statuses.

Problem: When defining state indicators, specifiers sometimes define types which are implementation-oriented, and are at once too general and too restrictive. In particular, a free type definition of $BOOLEAN$[1] is often overloaded (used to represent too many different kinds of statuses) and unhelpful (it cannot be used as a predicate, and cannot be extended to take further values).

[1] Z does not have a built-in Boolean type; predicates are either *true* or *false*.

Example:

$BOOLEAN ::= btrue \mid bfalse$

$switchon : BOOLEAN$

if $switchon = btrue$ then ...

Solution Choice:

1. Modelling membership or flags : free type
   Define each set of related statuses as a free type, with each status being represented by a branch with a meaningful name. The number of branches can be extended as more statuses are uncovered.
2. Modelling membership or flags : boolean schemas
   Use the schemas $True == [\ \mid true\ ]$; $False == [\ \mid false\ ]$ as predicates, and define $Boolean == \{True, False\}$.

Illustration 1:

$SWITCH ::= on \mid off$

$switch : SWITCH$

if $switch = on$ then ...

Note that this representation can easily be extended, for example, to

$SWITCH ::= on \mid off \mid broken$

Illustration 2:

$switch : Boolean$

if $switch$ then ...

Related patterns: A *Boolean* type should be used with caution; it may signal the Boolean flag antipattern, indicating the specification is insufficiently abstract.

■

8.3 Structural antipatterns

Fortran

Intent: Be as abstract as possible, avoiding over-specification of details such as algorithms

Problem: "Fortran programs can be written in any language"

■

Sørensen shorty

Intent: Use the Mathematical Toolkit to help write compact definitions, but don't be too clever[2]

Example: The ZRM Mathematical Toolkit definition of *squash* is

$$\forall f : \mathbb{N}_1 \pfun X \bullet \\ \quad squash\ f = f \circ (\ \mu\, p : 1 \upto \#f \rightarrowtail\!\!\!\!\!\twoheadrightarrow \mathrm{dom}\, f \mid p \circ succ \circ p^{\sim} \subseteq (\_ < \_)\ )$$

Illustration: The ISO-Z Mathematical Toolkit definition of *squash* is

$$\forall f : \mathbb{Z} \pfun X \bullet \\ \quad squash\ f = \{\ p : f \bullet \#\{\ i : \mathrm{dom}\, f \mid i \leq p.1\ \} \mapsto p.2\ \}$$

■

Boolean flag

Intent: Be specific about toggle- and selector-values: avoid the use of Boolean-valued variables.

■

[2] Named in honour of one of the pioneers of Z, Ib Sørensen, famed for his concise definitions. Name used with permission!

Chapter 9

Architecture patterns

9.1 Introduction

Architecture patterns capture conventional ways of using the formal language to produce an overall architecture that achieves the goals of the developer. Some architecture patterns draw on conventional wisdom in the construction of large or complex computer programs: such patterns are generalisable to other formal notations. Other patterns relate specifically to Z.

Architecture patterns help to express a description in Z. They also help the reader of the text. For example, the recognition that a particular form of expression is an architecture pattern allows the reader to concentrate on the system specific detail rather than the structure of the Z – this would apply to, for example, Delta/Xi : change part of the state, Delta/Xi : project away clutter, and promotion, among others.

Even experienced formalists find it difficult to take on a new style of specification. This can result in inappropriate use of common architecture patterns, and inelegant specifications that do not map cleanly onto the solution architecture. The Delta/Xi pattern [Barden *et al.* 1994, chapter 3] is widely (too widely?) used.

9.2 The architecture patterns

Morph

Intent: Specify the translation of one formal entity into another.

Problem: Some problems are cast in terms of one entity being translated into another, where this translation is independent of any hidden state – that is, the translation always provides the same result.

Example: A compiler translates source code into target code; it translates source code into error messages during syntax and static analyses.

Solution: Specify a function (or relation) that translates the first entity into the second. Use the structure of the entity being translated to break the function definition into named meaningful chunks. (In particular, if the entity being translated is of free type, break the function definition into the free type branches.)

Constraint: Any variation in the translation must be encoded as a parameter of the function, or emerge as the result of a non-deterministic relation. The translation function can be used within a Delta/Xi state or operation schema, with state variables being used as arguments, to provide such variation.

Related Patterns: Assemble from chunks, for defining a complicated or a multi-part translation. Modelling 1:many relations – if morph is providing the steps to be combined in a more general Unix-style pipe-and-filter specification, then make the relational choice.

Specimens: [Stepney 1993] specifies a compiler using meaning functions to define the high- and low-level languages, and a compilation function to define the translation.

■

Morph : diagram the structure

Intent: Summarise the structure of a Morph specification using a diagram.

Problem: Since a Z specification is presented 'bottom-up' (declaration before use) and can be factored into many pieces, it may become difficult to 'see the wood for the trees'.

Solution: Construct a diagram to record the structure of the functions, highlighting how their inputs and outputs are related.

Do not worry over-much about being consistent and complete, and about distinguishing every small difference: the purpose of the diagram is to be a graphical overview of structure, not to be an alternative formal notation.

The following components are recommended:

- draw the function being diagrammed as a grey background rectangle
- draw named functions used in the definition as named rectangles
- draw other functions, or parts of the specification that may be considered as functions, as named ovals (these may indicate refactoring opportunities)
- draw inputs as arrows into the box; if the input is a constant, draw it emanating from a constant source (indicated by a blob)
- draw outputs as arrows out of the box if the output is not used, draw it falling into a sink (indicated by a blob)
- label lines with any intermediate names introduced in the specification, as appropriate
- use different line styles to highlight different argument types
- 'explode' a product type input into its components (indicated by an 'exploding star' symbol)
- copy ('fork') an argument into several different functions (indicated by a 'plus' symbol)
- draw curried functions as nested functions, if necessary

As much as possible, without distorting the diagram, let the arguments flow from left to right. First minimise the number of crossing lines, and then as far as possible have the arguments entering the boxes in the right order.

Illustration: Figures A.4, A.5 show the morph diagrams (including the components listed above) of a large specification: two functions in a compiler specification.

■

Event traces

Intent: Specify an event-based system in terms of traces (with or without additional details of state and operations).

Problem: A system that is described in terms of possible events and their allowed sequences is not primarily composed of state and operations, so it is difficult to capture in a Delta/Xi model.

Solution: Specify the system using an event trace model. Specify the events, and describe the system as the set of traces (sequences of events) that are allowed or

disallowed.

Illustration:

- The system comprises events, with no elaboration.

  $[EVENT]$

- There are three separate types of event:

  $a, b, c : EVENT$

- The $SYSTEM$ comprises all event traces that have fewer a events than c events

  $$SYSTEM == \{\ s : \operatorname{seq} EVENT \mid \#(s \upharpoonright \{a\}) < \#(s \upharpoonright \{c\})\ \}$$

Constraints:

1. If modelling infinite sequences of events, the definitions in the Mathematical Toolkit needs supplementing: it defines only finite sequences.
2. A more complex model is needed if properties such as fairness or timing must be enforced. For pure event-trace systems, other formalisms such as CSP[Hoare 1985] are more appropriate than Z.

Variant: An event trace model can be combined with, or refined into, a Delta/Xi model where operations are used to add detail of the events. If the system is defined in terms of constraints on individual traces (as in the illustration), the trace can be included as a state component, with each operation adding the relevant event to the trace and obeying the constraint. However, if the system is defined in terms of properties across traces, much machinery is needed to do the combination formally [Cooper & Stepney 2000].

Related Patterns: Assemble from chunks, for defining a complicated set of sequences.

Specimens: [Cooper & Stepney 2000] specifies a segregation property as a property of the system's traces; [Heisel & Souquiéres 1999] captures system requirements as a trace model.

■

Delta/Xi

Intent: Specify a system characterised by discrete state elements and the operations on those states.

Solution: We do not go into details here. The style, sometime also called "State and Operations style", appears in most Z texts and papers, for example [Barden *et al.* 1994, chapter 3] (where it is called "the Established Strategy"). Its two main sub-patterns are

- Delta/Xi : state
- Delta/Xi : operations

Some more of its sub-patterns are given below.

Technical note: the so-called 'Delta/Xi naming conventions' are not pure *conventions*, because some semantics are defined. The dash (after-state of a variable) has semantics through schema composition; ? and ! (input and output variables) have semantics through ZRM's pipe, and ISO-Z gives a default semantics for the Δ and Ξ schema names (and is motivation for the Delta/Xi : strict convention sub-pattern).

Related Patterns: Name consistently when using the conventions of the style. Change part of the state when specifying which part of the state the operations change. Promotion allows local state and operations to be promoted to global state and operations. The partial precondition antipattern reveals a common error in the Delta/Xi total operation approach: a failure to cover all preconditions in the full operation.

■

Delta/Xi : disjoin errors

Intent: Structure the operations in terms of correct and erroneous behaviour.

Problem: An operation may behave normally only under a small precondition. Covering all the exception cases can lead to a large, unwieldy definition that is hard to understand.

Solution: Construct operation schemas for each correct and incorrect invocation.

Combine them using schema calculus disjunction:

$$FullOp == Op1Ok \lor Op2Ok \lor Op1Error \lor Op2Error \lor OtherErrors$$

Note that this construction can introduce non-determinism into the specification, where preconditions of the disjuncts overlap.

■

Delta/Xi : strict convention

Intent: Use the Δ/Ξ naming convention strictly, to avoid surprising the reader with hidden constraints.

Problem: If a schema such as ΔS is encountered, it usually has a conventional default meaning, $S \land S'$, but it may have been defined to mean something else (say an extra predicate or state component added). This can lead to confusion for the reader expecting the default meaning.

Solution: Adhere strictly to the convention. ΔS always means $S \land S'$; ΞS always means $[\ \Delta S \mid \theta S = \theta S'\ ]$. Consequently, it is never necessary to explicitly define ΔS or ΞS, as they always have their default meanings.

■

Delta/Xi : change part of the state

Intent: Define an operation that changes only a small part of the state.

Problem: In a Delta/Xi specification with a large state, S, most operations specify changes to only part of that state. ΞS constrains the entire state to be unchanged – it is usually too strong. ΔS puts no constraints on the state change – it is usually too weak.

Solution: Include a predicate of the form $\Xi S \setminus (x, y)$ in the operation schema, which constrains the state components not in the hiding list to be unchanged.

Variants:

1. Unlike ISO-Z, ZRM does not allow schema expressions to be used as predicates. In ZRM this pattern can be used by defining a new schema as the required hiding, and using a reference to the new schema as a predicate.

2. If the full state has been partitioned into Goldilocks chunks sub-states, the unchanging sub-states can be expressed as $\Xi SubS$, and the changing sub-state by ΔS. If the partition is done cleanly, most operations affect only a few sub-states; change part of the state is used on these sub-states.

Related patterns: Name predicates describes using a schema as a predicate. Delta/Xi : strict convention says why extra constraints should not be built into the ΔS schema

■

Delta/Xi : project away clutter

Intent: Retain a clear signature when introducing auxiliary variables to a Z schema.

Problem: In a Delta/Xi specification, local variables can be introduced by existential quantification in the predicate part, but that makes the predicate difficult to read, and the variables are not then available to other schemas if the definition is being assembled from chunks. Alternatively, local variables can be introduced in the declaration part, but then they are visible in the final schema, so are not strictly local, pollute the name-space, and potentially cause type problems.

Example:

$$
\begin{aligned}
&T1 == [\ S;\ x : X \mid predX\ ] \\
&T2 == [\ S;\ x : X;\ y : Y \mid predXY\ ] \\
&S0 == T1 \land T2
\end{aligned}
$$

has the x and y visible in $S0$'s signature, whereas

$$
\begin{aligned}
&T1 == [\ S \mid \exists\, x : X \bullet predX\ ] \\
&T2 == [\ S \mid \exists\, x : X;\ y : Y \bullet predXY\ ] \\
&S0 == T1 \land T2
\end{aligned}
$$

has the right signature for $S0$, but does not merge the x in $T1$ with that in $T2$.

Solution: Introduce the local variables in the declaration part of the relevant schema(s). Use Z projection, $\upharpoonright$, to list only the elements that must be visible.

Illustration:

$$T1 == [\ S;\ x : X \mid predX\ ]$$
$$T2 == [\ S;\ x : X;\ y : Y \mid predXY\ ]$$
$$Sfin == (T1 \land T2) \upharpoonright S$$

- *sfin* has all the variables of $T1$ and $T2$, with appropriate merges, but only the declarations of S are visible.

■

Delta/Xi : hide a state component

Intent: Take a state component out of the visible signature of a schema. (Z hiding operator.)

■

Delta/Xi : partial precondition

Intent: Write total operations.

Problem: Often, when using the Delta/Xi pattern, operations are intended to be total, but are mistakenly partial, because some case has been accidentally omitted.

Solution: Check that the operation is total, by calculating the precondition. Add any further necessary cases in the predicate part.

■

Object orientation (choice)

Intent: Specify the system in an object-oriented style.

Problem: Object-oriented structuring is appropriate to a development, yet there seems to be no natural way to use such a style in conventional formal methods such as Z.

Solution Choice: Depending on how much of the object-oriented machinery is need, several approaches are possible.

1. Object orientation : promotion
   Use promotion as a simple way of composing component parts into a whole.
2. Object orientation : Hall's style
   Use Hall's style [Hall 1990] to take promotion closer to an object-oriented style.
3. Object orientation : Object-Z
   If full-blown object orientation is required, use one of the object-oriented extensions to Z, such as Object-Z [Duke & Rose 2000], [Smith 2000].

Specimens: See the Object-Z web site at http://svrc.it.uq.edu.au/

■

Algebraic style

Intent: Specify a system in terms of properties.

Specimen: [Woodcock & Loomes 1988, chapter 11] presents a theory of natural numbers in an algebraic style. For large algebraic specifications, other formalisms may be more appropriate (for example, OBJ [Goguen & Malcolm 1996]).

■

Goldilocks chunks

Intent: Construct a large specification from substates that are "just the right size".

■

9.3 Architecture antipatterns

The only architecture antipattern here, consistent with our secondary aim of making the various styles more accessible to the developer, is:

Unsuitable Delta/Xi pattern

Intent: Use the architectural style most suited to the problem structure.

Problem: Because the Delta/Xi pattern is so common, sometimes specifiers try to use it for every specification, rather than only when it fits the problem structure.

Solution: Be aware that there are other specification structures (including those in this pattern catalogue), and use the most appropriate, or develop your own.

■

Chapter 10

Domain patterns

10.1 Introduction

Almost all high-level programming languages use class libraries and packages to provide incidental utilities and domain-specific concepts. In Z, toolkits play a similar role. The Mathematical Toolkit, which supplements core ISO-Z and ZRM, provides proven generic definitions, laws and constructs for use with sets, relations and predicates. However, its aim is to provide a sufficient set of laws, rather than a complete language.

We envisage toolkit libraries as a future parallel to pattern development in the engineering support for Z-based formal methods.

Toolkits could range in level and sophistication, from straightforward (but perhaps unproven) extensions to the Mathematical Toolkit, to complete application-specific support with template structures and special operators. Here, one small sample pattern is described in general terms. Others are represented collectively as the pattern "concept", application-oriented theory.

10.2 The domain patterns

Schema operator toolkit

Intent: Collect together schema operators for a specialised application.

Problem: A particular specification domain or style repeatedly combines schemas in a specific way, but there is no built-in schema operator that captures this com-

bination. A classic example is the *xor*, exclusive $\vee$ operator.

Solution: Use generics, and the 'type constrained generic' approach, to add a toolkit of user-defined schema operators.

Constraint: Most current Z tools do not support this extension.

Specimens: Type-constrained generics are described in [Valentine *et al.* 2000], along with examples of their use. Tools such as CADiℤ provide extensions to ISO-Z (or ZRM), largely in response to user demand.

■

Application-oriented theory

Intent: Structure the specification using concepts applicable to the domain.

Problem: Z is a powerful language, and as such, has little that is specific to any particular domain. It can be difficult to know where to start when specifying a new area.

Solution: Build suitable general definitions and conjectures for the application domain, and use them to help structure the specification and proofs. As experience in the domain grows, refactor these definitions into more generic forms, and make them the basis of a new "domain specific toolkit".

Specimens: [Woodcock & Loomes 1988, chapter 10] introduces this concept, illustrating it with a "revolutionary theory" of clocks. A few other specialist Z toolkits have been developed. A real number toolkit is described in the [Valentine 1993][Barden *et al.* 1994, chapter 10]. A fuzzy logic toolkit is presented in [Matthews & Swatman 2000]. Other toolkits, for infinite sequences, for trees and graphs, etc, have been suggested.

■

Chapter 11

Development patterns

11.1 Introduction

Development patterns give an engineering context for formal methods. Patterns can be written to assist the choice of whether to use a formal method, in the choice of a specific formal method, and in guiding the (top level) style of formality.

It is easy to write gibberish in a formal method. There are various ways during the development process of helping to validate a specification: to ensure that it is well-formed, meaningful, and says what is intended. These are captured as usage patterns.

Two development patterns have already been sketched:

- Convert to provable Z: see section 3.6.
- Convert to implementable Z: see section 3.6.

11.2 The development patterns

Focus the formality

Intent: Exploit mathematical rigour in an focussed way; apply it to only parts of the system or the development process.

Example: A system that is too large to develop efficiently with formal methods, or which does not justify the expenditure and personnel required for a formal development, nevertheless could benefit from some formality.

Solution: Decide where to focus the formal methods effort, based on which parts of the system are most critical, or least understood. In an iterative development, the focus can be widened in later iterations as more effort becomes justified, or more experience becomes available. This may require *refactoring* of earlier specifications.

■

Focus the formality (elaboration) : lightweight

Intent: Exploit formality in an *ad hoc* or loosely controlled way; demonstrate or detail only key parts of a system.

Problem: A system is being developed by a (set of) recognised non-formal methods or processes. The context requires or dictates this, but also requires that the level of confidence in certain properties is greater than can be guaranteed by the overall development approach.

Example: A system that has a small critical section, or a small number of critical properties that must be clearly specified and/or guaranteed.

Solution: Proceed with the informal specification as planned. Identify the critical section, or the part of the specification that is needed to investigate the critical properties. Using guidelines or intuition, produce a formal representation of the critical section and critical properties; produce some form of formal or informal proof of the required properties.

Constraint: The formal analysis is constrained (implicitly) by the assumptions of the formalisation; if a provably-correct Z model is needed for future development, there is unlikely to be a satisfactory lightweight specification.

Related patterns: Integrated methods, particularly those in which formal specifications are produced by guidelines rather than formal translation among formally-defined notations. Formal requirements, ie a formalisation of informal models for checking requirements, identifying missing constraints etc. Prove formally, and other validation approaches such as prove rigorously and animate.

Specimen: A lightweight Z analysis of CORBA [Basin *et al.* 2002]. One of the first uses of the term 'lightweight formalism' was in [Easterbrook *et al.* 1996], on spacecraft fault protection systems, using diagrammatic methods and PVS.

■

Focus the formality (elaboration) : requirements

Intent: Use a formal approach in the requirements elicitation process.

■

Focus the formality (elaboration) : specification only

Intent: Use Z to express or explore the specification, with no intention of attempting proof or formal development.

■

Do a refinement

Intent: Connect, by formal proof of a refinement conjecture, specifications of the same system at different levels of abstraction.

The generative patterns of classic refinements are,

- Do a refinement: abstract model
  Intent: Provide an abstract specification.
  ■
- Do a refinement: concrete model
  Intent: Provide an equivalent specification at a lower level of detail.
  ■
- Do a refinement: retrieve relation
  Intent: Formally express the mapping between each abstract component and its concrete equivalent.
  ■
- Do a refinement: implementation conjecture
  Intent: Prove that the concrete operations implement the abstract specification.
  ■

The underlying structure of a refinement is summarised in figure 11.1. The retrieve relation is the schema R. The refinement conjecture is represented as the ellipse, *refn*.

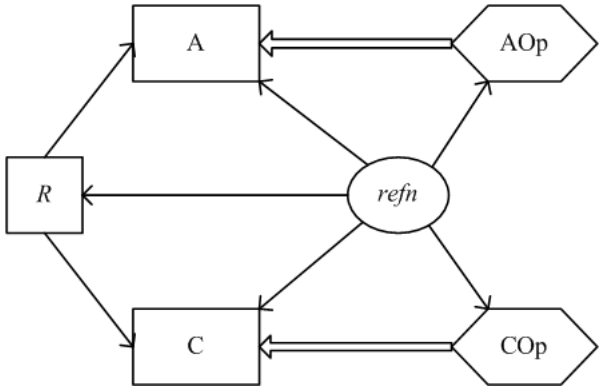

Figure 11.1 Structure of the refinement pattern using diagram the structure.

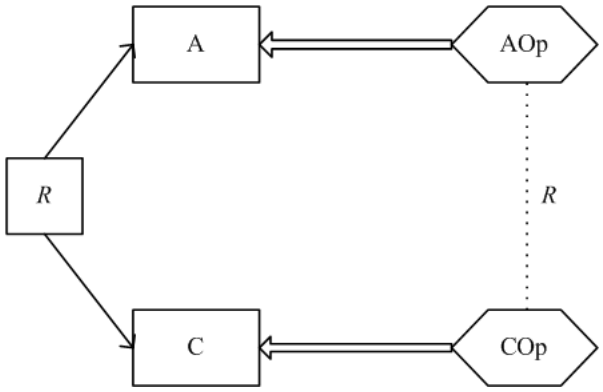

Figure 11.2 Structure of the refinement pattern, abbreviated form.

The conjecture statement refers to all states and operations. To clarify the structure, we choose to replace it by a dotted line linking the corresponding concrete and global operations referenced in the conjecture, and labelled with the relevant retrieve relation, figure 11.2.

The refinement pattern has a large number of elaborations, some of which are listed in table 12.1, below.

■

Use integrated methods

Intent: Use complementary formal methods, or formal and semi-formal methods, in an integrated manner

Specimens: [Semmens *et al.* 1992] is a summary of different formal and semi-formal combinations. Z and SSADM are used together by [Polack *et al.* 1993, Mander & Polack 1995]. Circus [Woodcock & Cavalcanti 2002] is a formal language that combines Z and CSP.

■

Prove rigorously

Intent: Establish a theorem by a combination of rigorous proof and informal argument appealing to intuition.

■

Prove formally

Intent: Formally prove a conjecture using mathematical laws and axioms of the appropriate domain.

■

Apply syntax and type checks

Intent: Find typographic errors and other accidental features of the formal text.

Related patterns: The *overloaded numbers* antipattern reduces the effectiveness of typechecking.

■

Animate

Intent: Explore dynamic properties of a specification. In particular: in a Delta/Xi specification, explore the sequence of states the system may progress through; in a Morph specification, explore the intermediate forms an entity takes during transformation.

■

Do sanity checks

Intent: Avoid common pitfalls and omissions in the formal and text descriptions.

Solution: Prove that the specification is not so constraining that it has no models. For Delta/Xi, prove the existence of the initial state and the non-emptiness of the precondition of every operation. Prove that the specification is not so loose that it permits undesired behaviour. For Delta/Xi, prove the totality and determinism of every operation, if such properties are required.

■

Express implicit properties

Intent: Make desired but implicit properties explicit.

Solution: Cast the desired properties as conjectures and prove rigorously or prove formally that they are consequences of the specification.

■

Chapter 12

Z generative patterns

Generative patterns are appropriate for any Z concept that is expressed in a series of steps or components. Simple generative patterns could be used to initiate beginners into the writing of specifications in any format.

At the hard end of formal notations, generative patterns are proposed to assist in refinement, retrenchment and proof.

Elaborations exist for all the proposed generative patterns. For example, the refinement pattern outlined above has elaborations to deal with particularly problematic elements of practical refinements. Patterns can be used to determine the kind(s) of refinement to use: forward or backward rules, blocking or non-blocking systems etc. In addition, there are elaborations that help the specifier to arrive at refinements. These could be used to guide the weakening of preconditions, and the making deterministic of the abstract specification (or their inverses for concrete-to-abstract refinement).

Table 12.1 identifies some complex Z patterns for which generative patterns have been identified. The list is not exhaustive, either in terms of the list of generative patterns, or in the detail of elaborations and component patterns.

Generative Pattern	**Elaboration**	**Intention**
`Delta/Xi:` state operations disjoin errors diagram the structure strict convention change part of the state project away clutter hide a state component partial precondition		Specify a system as a state, and with operations based on that state.
	`Promotion`	(see below)
`Promotion:` local state and operations global state framing schemas global operations diagram the structure		specify a system characterised as a collection of local states, and of operations based on those defined on the local state
	global constraints	add global state components to the collection of local instances
	internal identifiers	Use a native element of the local instances as the identifier
	combine promotions	specify a system that conforms to the local-global format for the Promotion structural pattern, but has different sorts of local instance
	multi-promotion	specify a system that comprises multiple instances, but has global operations that may affect more than one local instance
`Refinement` abstract model concrete model retrieve relation implementation conjecture		reduce the level of abstraction by provable refinements
	weakest concrete form	Use the retrieve relation to calculate the weakest concrete form.
	widen precondition	(Various patterns)
	reduced non-determinism	(Various patterns)
	backwards refinement	use backwards refinement rules
	blocking semantics	allow a blocking semantics in concrete specification

Table 12.1 Generative Z Patterns and their Elaborations

Part II

Refactoring

Chapter 13

Refactoring

Refactoring [Fowler 1999] originated as a technique for improving the structure of code, in a disciplined, manageable manner. The behaviour of the code remains unchanged, so refactorings can be validated by testing: the regression test suite is rerun after every meaning-preserving refactoring, so that any new bugs can be immediately detected and removed. A refactoring is never finalised until it has been shown to pass the tests; it is also critical that the original code is preserved until the refactoring process is shown to provide structural benefits – it must be possible to reverse the refactoring if necessary.

Refactoring is particularly emphasised as part of the XP discipline [Beck 1997], which relies heavily on an incremental approach to design and coding. The Cleanroom development process [Dyer 1992], which allows a component to be changed provided the input to output function is unchanged, can be thought of as an early example of meaning-preserving refactoring.

Refactorings can be used to ensure that evolving code makes use of patterns. Furthermore, the need or potential to refactor can be identified either reviewing code for antipatterns (Fowler's *bad smells*!) or by recognising sections of code as patterns, or by observing that there are patterns whose intent matches the overall intention of the (sub)program.

Each individual refactoring change is very small – renaming a component, or moving a feature, or splitting a method. Thus each change can be understood and tested in isolation. Large improvements to the structure of the code are made by applying a sequence of such small, controlled changes towards some goal.

Chapter 14

Refactoring in Z

14.1 Introduction

Like code, a specification is a model of a system. It can be refactored to improve its structure in many ways. Changes already encountered in this report include,

- changes to an existing specification as requirements evolve; refactoring may also be needed in order to extend the specification
- a specification written for one purpose (eg readability) may need to be re-structured for other purposes (eg to ease the proof of conjectures)
- a deeper appreciation of implementation constraints, requiring changes to the structure of the specification

As in code, refactoring can be used to bring a loosely-structured specification in line with particular structural and presentation patterns. Refactoring can also be used to convert between patterns; in particular, generative patterns can guide refactoring.

In formal development, refactoring is not confined to specification models. The same general concepts can be applied to proofs. Mathematicians invest effort in rewriting proofs to make them clearer, or more general, or shorter (or to find new insights in, and applications for, their theorems).

It may be essential to be able to refactor proofs, to make them more comprehensible to third parties, say. It may also be essential to be able to refactor proof scripts, the 'programs' that proof tools run to perform a proof. For example, changes to a specification may require that conjectures are re-expressed; re-proof could be made much more efficient if the existing proof script were refactored and re-run, rather than re-created from scratch. This is analogous to regression testing where code refactorings require modifications to the test suite. Proof refactoring is subject to

ongoing work, and is not considered further in this report.

14.2 Meaning preservation in Z

Refactoring preserves the meaning of a document. Meaning preservation is (theoretically) simple in code refactoring: the program must perform the same functions and produce the same outcomes from the same inputs. The notion is refined somewhat in the concept of bisimilarity, used to describe two transition systems that simulate each other (and adopted in, for example, comparing graphs)[1].

The meaning of a *specification* can be defined in many ways, making meaning-preservation in specification a rather more interesting concept than in programming.

Some possible variants of meaning preservation are listed below, with comments on their relevance and demonstrability.

- **model equality** : The expressions in two specifications should be semantically equivalent. Model equivalence could be established in the context of particular semantics, such as that established by ISO-Z [ISO-Z 2002]. The semantic equivalence of terms is established in the toolkits and standards; meaning preservation can be demonstrated if the specification changes use only equivalent terms. This form of meaning preservation is (should be) fully supportable by Z tools.
- **model isomorphism** : The expressions in two specifications should be semantically isomorphic (as in schema declarations and cross-products). This form of meaning preservation is (should be) fully supportable by Z tools.
- **observational equality** : The observed behaviour must be equal (for example, same inputs give same outputs, for Delta/Xi, or same traces for event traces) even though the preconditions may have been modified (ie one specification has a weaker operation precondition), and extra names may have been introduced.
- **observational isomorphism** : The observed behaviour must be isomorphic (allowing renaming, or refinement).
- **conjectural equivalence** : Provable properties should be preserved. Theorems on the original specification must be theorems of the refactored specifi-

[1] A transition system simulates another if it performs the same sequence of actions.

cation, and this must be established by posing equivalent conjectures (equivalent, because there may have been renaming, for example) and proving them true.

- **essential meaning equivalence** : The *essential meaning* [France 2001] of the specification is unchanged by the refactoring, in that the refactored specification still captures the intent and covers the same names of interest as the original specification. This weak definition can only be established to hold informally, for example by reference to domain experts. However, it should not be dismissed, particularly where refactoring takes place before the requirements are frozen.

There are undoubtedly other useful forms of meaning preservation in specification refactoring. There are also changes that are desirable but definitely change the meaning. These changes look superficially like refactoring steps. Such *benefactorings* can be used to fix bugs, to tidy up infelicities, and to upgrade specifications in a manageable way. This recognises that the purpose of changing a specification may be to modify its meaning, particularly early in a development, or in a maintenance revision.

Separating out refactorings (changing structure without changing meaning) from benefactorings (changing meaning without changing structure) can help provide a disciplined framework to specification evolution, and to the maintenance and upgrade process.

14.3 Incremental refactoring process

Each refactoring step should be the smallest logically complete change that can be made, and pushed all the way through the specification and proof. A small change to the specification can have a large knock-on effect on the proof. This is another reason refactoring steps should be as small as possible. It is very easy to get lost in the morass of changes, and forget or miss a needed change to a proof. Errors are easier to locate if the change is small. Small steps are also easier to document, and to record in the change lists.

For example, one of LFM's specifications has a system state comprising about 30 components, with about 15 predicates constraining them. Early on in the original specification development, the state was partitioned into four fairly independent sub-states (independent because few of the predicates involve components in differ-

ent substates, and most of the operations change components in a single sub-state). During one subsequent upgrade cycle, it became clear that one of the state components, and its associated predicates, would fit better in a different sub-state, because there it would result in significantly fewer of the predicates and operations referencing multiple substates. The declaration and the predicates were moved in a single refactoring step, and the required changes pushed through the operation definitions and proofs. This was the smallest logically complete step that could be made: taking two steps (declarations, then predicates, say) would have left the intermediate specification type-incorrect and unprovable.

Although a refactoring is the smallest possible complete change, some refactorings can be broken down into smaller steps that are only partly valid. Even though the whole specification and proof is inconsistent, portions of it can be checked. So, for example, in the context of a refinement proof, a modification can proceed by six small steps:

- modify and check the abstract specification
- modify and check proofs about the abstract specification
- modify and check the concrete specification
- modify and check proofs about the concrete specification
- modify and check the retrieve relation specification
- modify and check the refinement proof

At any step an error might be discovered (for example, that a global property does not hold). This might require earlier steps to be modified.

Benefactoring should use the same discipline as refactoring, of taking only small, provable (in some sense), controlled steps.

It is possible to introduce errors during an attempted refactoring step (especially at present, where tool support is not yet fully developed). Also, some refactoring steps can result in larger than anticipated changes to the specification, and particularly the proof. If the benefit from the refactoring is small, it might be better to revert rather than continue with the big restructuring. So make sure it is possible to roll back each change (using the version control system) so that it is possible to recover from mistaken attempts to refactor.

Refactoring can inflate change control lists, and make the changes look bigger than they really are, which might dismay any third party expecting to evaluate only a small upgrade. In LFM's projects, we took care to separate out the list of changes

that were refactorings from those that were real functionality changes, to help structure the evaluation task.

14.4 Identifying Z refactorings

The pattern language that we are constructing for Z allows us to capture insights as patterns; the patterns can then form steps or goals for refactoring. Refactoring opportunities become more apparent as the specifier gains experience. Additionally, performing proof gives additional insight into the structure of the system.

For example, in the electronic purse project [Stepney *et al.* 2000], it was noticed that certain combinations of state components appeared in the refinement proofs, and that these combinations mapped to meaningful concepts in the application domain. So these were introduced into the specification as derived state components. This refactoring improved the clarity and structure of both the specification and the proofs.

Lessons learned while doing the proof can be used to refactor the proof itself. For example, in the electronic purse project, a similar property was being proved several times. So this property was parameterised, extracted as a proved lemma, then used it several times in the main proof. This shortened and simplified the main proofs. The lemma made sense as a property in the application domain, so this refactoring also made the proof structure easier to understand.

Chapter 15

Refactoring to Promotion, by example

15.1 Introduction

This chapter describes refactoring to promotion by use of an example. We start with an initial specification closely based on the 'Abstract World' specification from the LFM electronic purse development[Stepney *et al.* 2000]. This is a case where a specification originally written in a flat Delta/Xi style is then identified later as a candidate complex enough for the promotion elaboration.

The structural promotion pattern is the target pattern for the refactoring, and the generative patterns guide the refactoring steps. Some of the refactoring transformations do not seem obvious and might be missed without the generative patterns as a reference. Indeed, some of the transformations temporarily make the specification structure more baroque.

15.2 Starting point: the existing specification

- purses are identified by a name

  $[NAME]$

- The abstract world comprises two mappings
  - *balance* maps purse names to their balances
  - *lost* maps purse names to the total amount they have lost because of failed transfer operations

AbWorld

$$balance : NAME \pfun \mathbb{N}$$
$$lost : NAME \pfun \mathbb{N}$$
$$\dom balance = \dom lost$$

- The *balance* and *lost* domains are the same, and, indeed define the purse names known to the system

There is a standard precondition for operations involving transfer between two purses. This is specified separately and included in each operation to aid readability, following the name predicates pattern.

- Transfer operations always have two input purse names and an input representing the value to be transferred between the purses.

AbTransferPre

$$AbWorld$$
$$from?, to? : NAME$$
$$value? : \mathbb{N}$$
$$\{from?, to?\} \subseteq \dom balance$$
$$to? \neq from?$$
$$value? \leq balance\ from?$$

- The two identified purses are known and distinct (it does not make sense to transfer money from one purse to itself).
- there is sufficient funds for the transfer.

There are two transfer operations; one that succeeds, and one that fails. They have the same declarations as *AbTransferPre*.

AbTransferOkay

$$\Delta AbWorld$$
$$from?, to? : NAME$$
$$value? : \mathbb{N}$$
$$AbTransferPre$$
$$balance' = balance \oplus \{from? \mapsto balance\ from? - value?,\ to? \mapsto balance\ to? + value?\}$$
$$lost' = lost$$

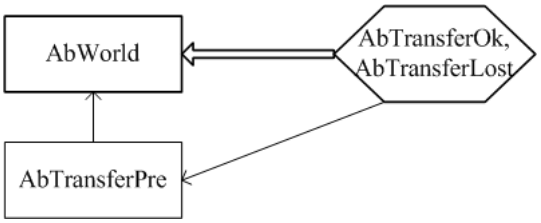

Figure 15.1 Structure of the abstract world of electronic purses, before refactoring

- A successful transfer decrements *value*? from the *from*? purse's balance, and adds it to the *to*? purse's balance, leaving all other balances unchanged.
- Success means the *lost* components are unchanged.

$$
\begin{array}{|l}
\hline
AbTransferLost \\
\Delta AbWorld \\
from?, to? : NAME \\
value? : \mathbb{N} \\
\hline
AbTransferPre \\
balance' = balance \oplus \{from? \mapsto balance\ from? - value?\} \\
lost' = lost \oplus \{from? \mapsto lost\ from? + value?\} \\
\hline
\end{array}
$$

- A lost transfer decrements *value*? from the *from*? purse's balance, and loses it, modelled by adding it to the *from*? purse's *lost* component.
- No other purse is affected.

Figure 15.1 represents the structure of the specification before refactoring. The Delta/Xi pattern is highlighted; there is also a schema that includes the state, used (as a precondition) in the operations.

This specification matches the promotion intent; it is a Delta/Xi specification, and it describes a system made up of instances of a local state and global operations based on operations on the local states.

The refactoring steps are based on the four elements of the generative pattern for promotion. The first generative pattern, local state and operations, requires substantial refactoring over several steps. The second generative pattern, global state, emerges as a side effect.

15.3 Step 1: introduce local state

The first goal for refactoring is to define the local state. The system described above has a collection of purses, so the purse becomes the local instance.

The individual purses each have a *balance* and a *lost* component. It is formed by removing the identifying functions and predicates from the original state specification. The result matches the state element of the local state and operations generative pattern element:

$$
\begin{array}{|l}
\hline
AbPurse \\
balance : \mathbb{N} \\
lost : \mathbb{N} \\
\hline
\end{array}
$$

The global elements from the original state are refactored to match the global state pattern:

$$[NAME]$$

$$
\begin{array}{|l}
\hline
AbWorld \\
abPurse : NAME \pfun AbPurse \\
\hline
\end{array}
$$

The preservation of meaning for *abPurse* can be established by expanding the schema to a single function from $NAME$ to the tuple $(balance, lost)$. This is equivalent to the original state schema, $AbWorld$.

To complete these refactorings, and to preserve the full sense of the original specification, the original precondition and operation specifications have to be refactored to use the local and global states.

- The precondition schema declarations are not affected: purses are still identified by name, even though the name has been relocated to the global context.

AbTransferPre

$$
\begin{array}{l}
AbWorld \\
from?, to? : NAME \\
value? : \mathbb{N} \\
\hline
\{from?, to?\} \subseteq \mathrm{dom}\ abPurse \\
to? \neq from? \\
value? \leq (abPurse\ from?).balance
\end{array}
$$

- Each reference to a purse in the original precondition is replaced by a call to or application of the *abPurse* function. Otherwise, the precondition predicates are unchanged.

The operations must introduce the global and local state schemas. To allow global operations to use the local state, explicit after-state purses are constructed.

- A successful transfer involves explicitly two purses. The refactoring is influenced by the form of the elaboration pattern, multi-promotion in its version for a fixed number of local instances. The declarations are made accordingly. Making a schema binding 2 is used to construct purse instances.

AbTransferOkay

$$
\begin{array}{l}
\Delta AbWorld \\
\Delta AbPurse_{\ 1} \\
\Delta AbPurse_{\ 2} \\
from?, to? : NAME \\
value? : \mathbb{N} \\
\hline
AbTransferPre \\
AbPurse'_{\ 1} = (\ \mu \Delta AbPurse \mid \theta AbPurse = abPurse\ from? \\
\qquad \land\ balance' = balance - value? \\
\qquad \land\ lost' = lost \\
\qquad \bullet\ AbPurse'\ ) \\
AbPurse'_{\ 2} = (\ \mu \Delta AbPurse \mid \theta AbPurse = abPurse\ to? \\
\qquad \land\ balance' = balance + value? \\
\qquad \land\ lost' = lost \\
\qquad \bullet\ AbPurse'\ ) \\
abPurse' = abPurse \oplus \{from? \mapsto \theta AbPurse'_{\ 1}, to? \mapsto \theta AbPurse'_{\ 2}\}
\end{array}
$$

- $AbPurse_{\ 1}$ has *value*? subtracted from its *balance*; no other element is changed.

- *AbPurse* $_2$ has *value*? added to its *balance*; no other element is changed.
- The global world mappings to the instances of the two purses are overwritten with the after-states of each purse.

The lost case is refactored similarly.

- An unsuccessful transfer involves exactly one purse. The refactoring is influenced by the form of the standard pattern, framing schemas. The declarations are made accordingly.

AbTransferLost

$$\Delta AbWorld$$
$$\Delta AbPurse$$
$$from?, to? : NAME$$
$$value? : \mathbb{N}$$

AbTransferPre

$$AbPurse' = (\ \mu \Delta AbPurse \mid \theta AbPurse = abPurse\ from? \land balance' = balance - value? \land lost' = lost + value? \bullet AbPurse'\ )$$
$$abPurse' = abPurse \oplus \{from? \mapsto \theta AbPurse'\}$$

- *AbPurse* has *value*? subtracted from its *balance*, and added to *lost*.
- The global world mapping to the purse is overwritten with the purse's after-state.

This intermediate refactoring preserves the meaning of the original specification, but is otherwise considerably more obscure than the original. Further refactoring is required[1].

15.4 Step 2: introduce local operations

Having extracted the local and global states, attention now turns to the extraction of local operations. These are specified to include the local part of the precondition schema.

[1] This first refactoring is, in fact, closer to the form of the actual AbPurse specification[Stepney *et al.* 2000]. See Appendix A.1 for diagrams of the full specification.

- The declaration of *AbWorld* has been replaced by the local *AbPurse*.
- The other declarations are again as for the original precondition schema.

TransferPre

$$
\begin{array}{l}
AbPurse \\
from?, to? : NAME \\
value? : \mathbb{N} \\
\hline
value? \leq balance \\
to? \neq from?
\end{array}
$$

- The predicates are the local ones from the original precondition schema.
- The validity of *from*? and *to*? cannot be checked in the local context.

The local operations separately specify transfer from a purse, transfer to a purse and the lost transfer. The schemas are taken to be self-explanatory.

TransferFrom

$$
\begin{array}{l}
\Delta AbPurse \\
from?, to? : NAME \\
value? : \mathbb{N} \\
\hline
TransferPre \\
balance' = balance - value? \\
lost' = lost
\end{array}
$$

TransferTo

$$
\begin{array}{l}
\Delta AbPurse \\
from?, to? : NAME \\
value? : \mathbb{N} \\
\hline
TransferPre \\
balance' = balance + value? \\
lost' = lost
\end{array}
$$

TransferLost

$\Delta AbPurse$
$from?, to? : NAME$
$value? : \mathbb{N}$

TransferPre

$balance' = balance - value?$
$lost' = lost + value?$

15.5 Step 3: introduce framing schemas

The next generative pattern is framing schemas. Each kind of operation requires a frame. All the local operations are simple changes to the local state. However, at the global level, a successful transfer updates the state of two purses whilst the unsuccessful transfer updates only one purse.

The validity of the purses involved in a transfer must be checked at the global level, since the name identifiers have only been assigned to the global state. This check needs to appear as part of the frame(s).

- The declarations are as for the refactored transfer operation above

$\Phi Transfer$

$\Delta AbWorld$
$\Delta AbPurse_1$
$\Delta AbPurse_2$
$from?, to? : NAME$
$value? : \mathbb{N}$

$\{from?, to?\} \subseteq \mathrm{dom}\ balance$
$from? \neq to?$
$\theta AbPurse_1 = abpurse\ from?$
$\theta AbPurse_2 = abpurse\ to?$
$abPurse' = abPurse \oplus \{from? \mapsto \theta AbPurse'_1, to? \mapsto \theta AbPurse'_2\}$

- The identities of the two purses are checked against the known identities.

- The *abPurse* after-state is formed by overriding the mappings for the two instances with the results of the local operations.

The single purse *TransferLost* could be provided with a framing schema, $\Phi\, TransferLost$, by an application of the framing schemas pattern:

$$
\begin{array}{|l}
\hline
\Phi\, TransferLost \\
\Delta AbWorld \\
\Delta AbPurse \\
from?, to? : NAME \\
value? : \mathbb{N} \\
\hline
\{from?, to?\} \subseteq \mathrm{dom}\ balance \\
from? \neq to? \\
\theta AbPurse = abpurse\ from? \\
abPurse' = abPurse \oplus \{from? \mapsto \theta AbPurse\ '\} \\
\hline
\end{array}
$$

However, it is equally valid, and requires fewer definitions, to think of the unsuccessful global operation as affecting two purses, one of which is unchanged.

15.6 Step 4: Define the global operations

The final step applies the global operations pattern.

$$AbTransferOkay == \exists \Delta AbPurse\ _1;\ \Delta AbPurse\ _2 \bullet \Phi Transfer \land TransferFrom\ _1 \land TransferTo\ _2$$

$$AbTransferLost == \exists \Delta AbPurse\ _1;\ \Delta AbPurse\ _2 \bullet \Phi Transfer \land TransferLost\ _1 \land \Xi AbPurse\ _2$$

Note: the version using the $\Phi\, TransferLost$ frame is:

$$AbTransferLost == \exists \Delta AbPurse \bullet \Phi\, TransferLost \land TransferLost$$

■

15.7 Resulting specification, summary

Gathering together all the pieces produced above, the complete specification of the abstract purse world comprises (a) local specifications, (b) framing schema(s) and (c) global transfer operations that promote the local transfer elements.

This could be demonstrated to be strictly equivalent to the original by a *schema expansion* refactoring of both versions, and simplifying. Alternatively, conjectures on the operations could be formulated and proved. Neither is demonstrated here.

Local state : one purse

$[NAME]$

AbPurse

$$balance : \mathbb{N}$$
$$lost : \mathbb{N}$$

TransferPre

$$AbPurse$$
$$from?, to? : NAME$$
$$value? : \mathbb{N}$$

$$value? \leq balance$$
$$to? \neq from?$$

TransferFrom

$$\Delta AbPurse$$
$$from?, to? : NAME$$
$$value? : \mathbb{N}$$

$$TransferPre$$
$$balance' = balance - value?$$
$$lost' = lost$$

TransferTo

$\Delta AbPurse$
$from?, to? : NAME$
$value? : \mathbb{N}$

TransferPre

$balance' = balance + value?$
$lost' = lost$

TransferLost

$\Delta AbPurse$
$from?, to? : NAME$
$value? : \mathbb{N}$

TransferPre

$balance' = balance - value?$
$lost' = lost + value?$

Global state : world of purses

AbWorld

$abPurse : NAME \pfun AbPurse$

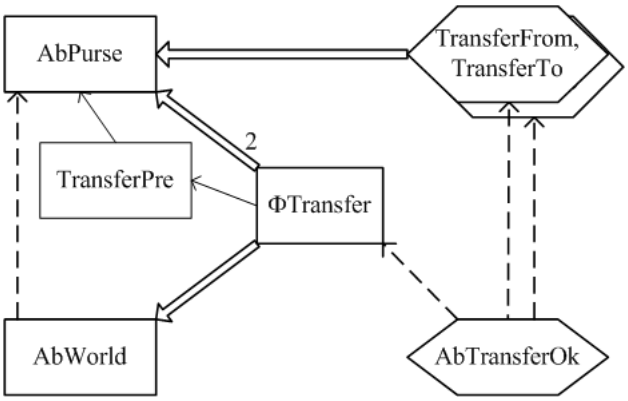

Figure 15.2 Structure of the abstract world of electronic purses, after refactoring to a two-state promotion

$$
\begin{array}{|l}
\hline
\Phi Transfer \\
\Delta AbWorld \\
\Delta AbPurse_{\ 1} \\
\Delta AbPurse_{\ 2} \\
from?, to? : NAME \\
value? : \mathbb{N} \\
\hline
\{from?, to?\} \subseteq \mathrm{dom}\ balance \\
from? \neq to? \\
\theta AbPurse_{\ 1} = abpurse\ from? \\
\theta AbPurse_{\ 2} = abpurse\ to? \\
abPurse' = abPurse \oplus \{from? \mapsto \theta AbPurse'_{\ 1}, to? \mapsto \theta AbPurse'_{\ 2}\} \\
\hline
\end{array}
$$

$$
\begin{array}{l}
AbTransferOkay == \exists \Delta AbPurse_{\ 1};\ \Delta AbPurse_{\ 2} \bullet \\
\qquad\qquad \Phi Transfer \wedge TransferFrom_{\ 1} \wedge TransferTo_{\ 2} \\
AbTransferLost == \exists \Delta AbPurse_1;\ \Delta AbPurse_{\ 2} \bullet \\
\qquad\qquad \Phi Transfer \wedge TransferLost_1 \wedge \Xi AbPurse_{\ 2}
\end{array}
$$

The specification is slightly longer, but more readable. Writing further operations that change purses is straightforward: the local component operations are added, and the global operation pattern applied.

Figures 15.2 represents the structure of the specification after refactoring. Fig-

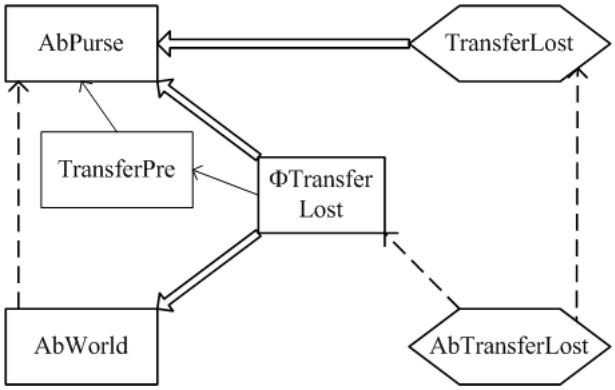

Figure 15.3 Structure of *AbTransferLost* after refactoring using the single local instance promotion

ure 15.3 shows the alternative form of *AbTransferLost*, using a separate, single-instance promotion. The diagrams look more complicated than the pre-refactoring diagram; this demonstrates how much specification structure was implicit before (hidden in the state and operation schemas). The more explicit structure makes a specification easier to comprehend.

Chapter 16

Refactoring catalogue

16.1 Introduction

In this chapter we describe some of the refactorings that LFM found useful in its various Z projects. Some of these are done simply to improve and clarify structure. Others are done to conform to various Z conventions, as expressed in [Barden *et al.* 1994], and in the Z patterns described above.

The proofs in these projects were all performed by hand, with minimal tool support (type checking only). The specifications and proofs were independently evaluated; some of the structural improvements are designed to make evaluation of proofs easier. Different refactorings might be more appropriate for tool-supported proofs (discussed below).

Many refactorings can be applied in either direction (since they are meaning preserving). In particular, every choice pattern has an associated refactoring, converting between the choices. Which direction to perform the refactoring in a particular case depends on the specific context.

16.2 Structure of Z refactoring descriptions

The statement of each refactoring has the following structure (some parts may be omitted for brevity)

- *Name*
- **Problem** : Short statement of the problem
- **Solution** : Shorter statement of the solution

- **Discussion** : Short statement of the refactoring change, and issues
- **Steps** : Process to be followed to achieve the refactoring
- **Example**
- **Variants**

16.3 Refactoring choice patterns

As we noted earlier, every choice pattern has an associated refactoring, converting between the choices. Some of the explicit refactorings are given here.

Convert a Cartesian Product to a Schema

Problem : You have a Cartesian product type being used as a record, with lots of component references.

Solution : Introduce a schema product type, and use it instead.

Discussion : This converts from a one use of the choice pattern, modelling product types : Cartesian product, to the other, modelling product types : schema. The components of a Cartesian product are labelled by their positions, a not-very meaningful number. The components of a schema product are labelled by their names, and these can be chosen to be much more meaningful.

Steps :

- create the new definition, with a new name (if the name of the new and old definitions are to be the same, first do a refactoring to rename the old definition)
- typecheck, to ensure there are no name clashes
- change each occurrence of the old name to the new one, and each occurrence of a component reference from the number to its name (this requires more than just mechanical changes, see the example)
- typecheck, to ensure the name has been used properly
- propagate the replacement through the proofs
- delete the old definition
- typecheck, to ensure no uses have been missed

Example : A syntactic structure is defined as a Cartesian product; its semantics are given by meaning functions applied to the numbered components.

$$binaryOp == EXPR \times OP \times EXPR$$

$$\forall\, b : binaryOp \bullet M_e\ b = M_o\ b.2\ (M_e\ b.1, M_e\ b.3)$$

The refactoring redefines the syntactic structure as a schema product; its semantics are given by meaning functions applied to the named components.

$$BinaryOp == [\ lhs, rhs : EXPR;\ op : OP\ ] \ldots$$

$$\forall\, BinaryOp \bullet M_e\ \theta BinaryOp = M_o\ op\ (M_e\ lhs, M_e\ rhs)$$

■

Curry a Function

Problem : You have a function argument that is a product type, but want to apply the function to only part of that product.

Solution : Replace the product type with a curried form.

Discussion : The currying pattern used here is a lower-level of the choice pattern, modelling product types.

Example : before

$$add : \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N}$$
$$increment : \mathbb{N} \rightarrow \mathbb{N}$$
$$\forall\, m, n : \mathbb{N} \bullet add(m, n) = m + n$$
$$\forall\, n : \mathbb{N} \bullet increment\ n = add(1, n)$$

After

$$add : \mathbb{N} \nrightarrow \mathbb{N} \rightarrow \mathbb{N}$$
$$increment : \mathbb{N} \rightarrow \mathbb{N}$$
$$\forall\, m, n : \mathbb{N} \bullet add\ m\ n = m + n$$
$$increment = add\ 1$$

■

16.4 Other refactorings

Rename a Component

Problem : You have a specification component with a name that does not indicate its purpose, or otherwise does not name consistently

Solution : Change the name.

Discussion : This refactoring is most useful in the early stages of specification. Initial names choices can become inappropriate as the specification develops and the purpose of a component is clarified. The refactoring may even be a response to the *overmeaningful name* antipattern, in which the initial name contains more "semantics" than its definition. The naming convention is often evolving at this point: name consistently may be applied only once the naming conventions have stabilised.

Steps :

- choose the new name
- update the naming convention documentation if necessary
- rename in the definition, and any indexes that refer to it
- propagate the name change throughout the specification and proofs.

A typechecker can help to find all the places the name needs to be changed.

■

Extract Commonality

Problem : You have a term, predicate or chunk that is used in several places in the specification or proof.

Solution : Introduce a new definition to name meaningful chunks or name predicates, and use that name in place of the term.

Discussion : Make sure the newly named term is a meaningful concept in its own right, not just derived from a textual coincidence. Let the name capture this meaning (and follow any naming convention pattern in use). The use of the name

makes the specification more readable, and more concise.

The new name might be introduced as a global definition (possibly an entry in a domain specific toolkit), a schema, a derived state component, a local definition (existentially quantified), or a lemma (in which case the "name" occurs in the informal commentary rather than the formal text).

Steps :

- create the new definition
- typecheck, to ensure there are no name clashes
- for each use of the term in the scope of the definition
  - replace the use of the term with the name
  - typecheck, to ensure the name has been used properly in this case
  - propagate the replacement through the proofs (this may require the addition of an expansion step, replacing the name with its definition, at each point in the proof when the definition is used)

Variants : A special case of this refactoring is *Genericise Common Definitions*, used where several similar definitions acting on different types. The solution is to define a generic construct that captures all the behaviours. This uses generics to control detail.

■

Inline a Name

Problem : You have a name with a relatively simple definition, used only once or a few times.

Solution : Remove the name, and replace its use(s) with its definition.

Discussion : This is essentially the Extract Commonality refactoring in reverse. Getting the right size of chunking is an art. Too few names and the reader has to puzzle out large swathes of mathematics. Too many names, and the reader has to remember their meaning when puzzling out their uses.

Steps :

- for each use of the name
  - replace the name with its definition

  - typecheck, to ensure the name has been used properly
- delete the definition of the name
- typecheck, to ensure no uses have been missed

■

Split a State Component

Problem : You have a state component that is a product type, where each component is being referenced independently of the others.

Solution : Replace the single component with a separate component for each part of the product.

Discussion : The parts of the product type are acting independently, and so do not need to be bundled together.

Example : Structure before:

$$S == [\ c : A \times B;\ \ldots \mid \mathcal{P}(c.1);\ \mathcal{Q}(c.2);\ \ldots\ ]$$

Structure after:

$$S == [\ ca : A;\ cb : B;\ \ldots \mid \mathcal{P}(ca);\ \mathcal{Q}(cb);\ \ldots\ ]$$

■

Merge State Components

Problem : You have two or more state components that are constantly being referenced together.

Solution : Replace the separate components with a single product type component (Cartesian or schema product).

Discussion : The separate components are acting as parts of a greater whole, and so can be combined into that whole.

Example : Structure before:

$$S == [\ ca : A;\ cb : B;\ \ldots \mid \mathcal{P}(ca, cb);\ \mathcal{Q}(ca, cb, \ldots);\ \ldots\ ]$$

Structure after (Cartesian product):

$$S == [\ c : A \times B;\ \ldots \mid \mathcal{P}(c);\ \mathcal{Q}(c, \ldots);\ \ldots\ ]$$

Structure after (schema product):

$$C == [\ ca : A;\ cb : B\ ]$$
$$S == [\ c : C;\ \ldots \mid \mathcal{P}(c);\ \mathcal{Q}(c, \ldots);\ \ldots\ ]$$

If some of the predicate part can also be bundled, alternative structure after (Cartesian product):

$$C == \{\ ca : A;\ cb : B \mid \mathcal{P}(ca, cb)\ \}$$
$$S == [\ c : C;\ \ldots \mid \mathcal{Q}(c, \ldots);\ \ldots\ ]$$

Alternative structure after (schema product):

$$C == [\ ca : A;\ cb : B;\ \ldots \mid \mathcal{P}(ca, cb)\ ]$$
$$S == [\ c : C;\ \ldots \mid \mathcal{Q}(c, \ldots);\ \ldots\ ]$$

■

Split a State into Substates

Problem : You have a large number of state components.

Solution : Structure the state into substates.

Discussion : This applies the name meaningful chunks and assemble from chunks patterns. Smaller chunks of state may be easier to understand, and may simplify operation definitions. Components that constrain each other, and components that change together, are candidates for being in the same substate. Most state predicates are on substates, with few on the global state. Also most operations affect only a single substate, and the unchanging nature of the other substates can be captured with a few Ξ schemas, rather than a long list of unchanging state components (see Delta/Xi : change part of the state).

This refactoring may be followed by *expand schemas slowly* refactoring, to take advantage of the opportunity to expand only the relevant substates (this may result in some extra steps being added, as the substates are expanded one by one).

Steps :

- define substate schemas; typecheck
- modify the state schema to use these; typecheck
- modify each operation schema to use these, including the use of Δ and Ξ substate schemas; typecheck
- define substate initialisation operations; typecheck
- use schema calculus to define the state initialisation operation using the substate initialisations; typecheck
- propagate through the proofs

Example : before

$$S == [\ x, y : \mathbb{Z};\ a, b : \mathbb{P}\,\mathbb{Z} \mid x \in a;\ y \notin b;\ a \neq b\ ]$$
$$Op == [\ \Delta S;\ x? : \mathbb{Z} \mid x' = x?;\ a' = a \cup \{x?\};\ y' = y;\ b' = b\ ]$$
$$InitS == [\ S' \mid x' = 0;\ y' = 1;\ a' = \{x'\};\ b' = \varnothing\ ]$$

After

$$Sx == [\ x : \mathbb{Z};\ a : \mathbb{P}\,\mathbb{Z} \mid x \in a\ ]$$
$$Sy == [\ y : \mathbb{Z};\ b : \mathbb{P}\,\mathbb{Z} \mid y \notin b\ ]$$
$$S == [\ Sx;\ Sy \mid a \neq b\ ]$$
$$Op == [\ \Delta Sx;\ \Xi Sy;\ x? : \mathbb{Z} \mid x' = x?;\ a' = a \cup \{x?\}\ ]$$
$$InitSx == [\ Sx' \mid x' = 0;\ a' = \{x'\}\ ]$$
$$InitSy == [\ Sy' \mid y' = 1;\ b' = \varnothing\ ]$$
$$InitS == InitSx \land InitSy \ldots$$

■

Move a State Component between Substates

Problem : You have a substate component frequently being used along with components in another substate.

Solution : Move the component and any predicates into the other substate. This normally has to be done as one step, since the specification is likely to inconsistent during the move. The specification is only typechecked after the whole move.

■

Split an Operation into Disjuncts

Problem : You have an operation with a top level disjunct amongst its predicates.

Solution : Split the operation into two parts, one for each disjunct.

Example : before

$$Op == [\ \Delta S \mid \mathcal{P};\ \mathcal{Q} \lor \mathcal{R};\ \mathcal{S}\ ]$$

After

$$OpQ == [\ \Delta S \mid \mathcal{P};\ \mathcal{Q};\ \mathcal{S}\ ]$$
$$OpR == [\ \Delta S \mid \mathcal{P};\ \mathcal{R};\ \mathcal{S}\ ]$$
$$Op == OpQ \lor OpR$$

■

Split an Operation into a Composition

Problem : You have an operation with some existentially quantified state components that are acting as "intermediate" variables.

Solution : Split the operation into two on these components, and compose the parts.

Discussion : This is effectively expanding out the definition of schema composition[1].

Example : before

$$Op == [\ \Delta S \mid \exists S_0 \bullet \mathcal{P}(\theta S, \theta S_0) \land \mathcal{Q}(\theta S_0, \theta S')\ ]$$

After

$$Op1 == [\ \Delta S \mid \mathcal{P}(\theta S, \theta S')\ ]$$
$$Op2 == [\ \Delta S \mid \mathcal{Q}(\theta S, \theta S')\ ]$$
$$Op == Op1 \fatsemi Op2$$

[1] This suggests there should be a similar refactoring for schema piping. However, since we have yet to come across a realistic case of piping in a large specification, we leave out the description.

■

Reorder Product Arguments

Problem : You have two functions, one with a product range, one with a product domain, that you want to compose, but the products have their items in different orders.

Solution : Reorder one of the products to match the other.

Example : before

$$\begin{array}{|l} f : X \pfun Y \times Z \\ g : Z \times Y \pfun W \\ h : X \pfun W \\ \hline \forall\, x : X \bullet \exists\, y : Y;\ z : Z \mid (y, z) = f\ x \bullet h\ x = g(z, y) \end{array}$$

After:

$$\begin{array}{|l} f : X \pfun Y \times Z \\ g : Y \times Z \pfun W \\ h : X \pfun W \\ \hline h = f \fatsemi g \end{array}$$

■

Reorder Curried Arguments

Problem : You have a function that you want to partially apply, but the argument order does not support that.

Solution : Reorder the arguments so that it can be partially applied.

Example : before

$$\begin{array}{|l} f : X \pfun Y \pfun Z \\ g : X \pfun Z \\ \hline \exists\, y : Y \bullet \forall\, x : X \bullet f\ x\ y = g\ x \end{array}$$

After

$$
\begin{array}{|l}
f : Y \pfun X \pfun Z \\
g : X \pfun Z \\
\hline
\exists\, y : Y \bullet f\ y = g
\end{array}
$$

■

Thin Early, Thin Often

Problem : You have a large, clumsy hypothesis in a goal you are trying to prove.

Solution : Thin the hypothesis as early and as often as is possible.

Discussion : Thinning declarations and predicates from the hypothesis has two advantages: it keeps the hypothesis small and manageable; and it indicates more clearly what remaining properties are still required to discharge the goal.

■

Move a Common Proof Step Before a Branch Point

Problem : You have a proof that has branched into several "cases" (for example, when splitting up a disjunct in the hypothesis), and a similar step is used in each branch (for example, cutting in a particular value).

Solution : Move the step before the case split, and do it only once.

Discussion : If the step is only "similar" in each branch, it is first necessary to apply some other refactorings to make the step the same in each branch (for example, by parameterising the proof on the case branch parameter) before this refactoring can be performed.

■

Turn a Common Proof Step into a Lemma

Problem : You have a chunk of proof that is used in several places.

Solution : Extract the commonality as a lemma.

Discussion : This is a special case of *extract commonality*. Make sure the lemma is meaningful in isolation, and is not just a "textual macro".

■

Schema expansion

Problem: You need to manipulate the internals of a schema, or schema expression.

Solution: Expand the schema.

Discussion: Sometimes it is necessary to present a schema in an expanded or fully normalised form, to understand the internals, or to negate a schema, or to calculate a precondition, for example. This refactoring should make use of the *Expand schemas slowly* refactoring.

Steps:

- Expand the schema calculus operators, using their definitions.
- Expand schema components, by inlining included schemas.
- Expand toolkit definitions in declarations, and move the constraints to the predicate part, leaving only type information.

■

Expand schemas slowly

Problem : You have a large or deeply nested schema that is being expanded, or flattened, in a single step, resulting in the sudden appearance of a lot of (as yet) unnecessary terms.

Solution : Expand the schema incrementally, exposing only those terms needed at any given stage.

Discussion : This may require the prior use of the split a state into substates or the split an operation into disjuncts refactoring, to provide parts that can be expanded separately.

If the schema is used as a predicate in the goal's hypothesis, consider duplicating the required term outside the schema, rather than expanding the schema. Use the term as required, then thin it.

■

16.5 Refactoring as a proof technique

The concept of refactoring can be used when performing a (hand) proof. A proof can be considered to be the documentation of a sequence of meaning preserving transformations of a goal that improves its structure in a particular way, to the predicate *true*.

Many proof steps can be applied in either a forward or backward manner, and hence are meaning preserving. Examples include one-pointing or applying Leibniz (replacing equals by equals), schema expansions, and various eliminations. Such steps can be treated as refactorings. The presentation of the proof can be constructed by repeating the following steps:

- copy and paste the most recent form of the goal
- perform the desired refactoring by suitably editing the copy
- typecheck

For brevity of presentation, several refactorings (such as several one-pointings) may be performed on a single copy of the goal. However, to make such a presentation step comprehensible to reviewers, no single part of the goal changed by a refactoring should be further changed by a subsequent refactoring in the same step (that is, restrict multiple refactorings to distinct parts of the goal).

The names of the refactorings performed form part of the documentation of the proof step.

Chapter 17

Benefactorings

Some benefactorings used in LFM projects are noted here.

Change a Type

Problem : A functionality change requires a component's type to be changed. For example, a simple type might need to be extended to a product type.

Solution : Before adding any new functionality, make the minimal change to the type (so, add the new component throughout, but don't use it yet).

Steps :

- modify the type in the abstract model; typecheck
- modify any global property proofs about the abstract model
- modify the type in the concrete model; typecheck
- modify any global property proofs about the concrete model
- modify the retrieve relation that links the modified types if appropriate; typecheck
- modify the refinement proof as necessary

■

Add a State Component

Problem : A functionality change requires a new state component.

Solution : Add the component, and perform the knock-on changes, without adding any extra functionality based on the component.

Discussion : This is the *change a type* refactoring applied in a slightly larger context to adding a state component to a Delta/Xi specification, in order to add more functionality.

Steps :

- add the component to the abstract model; typecheck
- add abstract constraints; typecheck
- modify any global property proofs about the abstract model
- add the component to the concrete model; typecheck
- add concrete constraints; typecheck
- modify any global property proofs about the concrete model
- modify the retrieve relation to link the new abstract and concrete components; typecheck
- modify the refinement proof as necessary

■

Add an Operation

Problem : A functionality change requires a new operation.

Solution : Add the operation, and perform the knock-on changes.

Discussion : When following the Delta/Xi pattern, adding an operation has little effect on the specification. If the model has theorems about global properties, it is necessary to modify their proofs to ensure the properties still hold.

Variant : Steps for promotion

- add the new local operation; typecheck
- add a new framing schema, if necessary
- add the new global operation; typecheck
- modify any global property proofs about the model

Variant : Steps for refinement

- add the operation to the abstract model; typecheck
- modify any global property proofs about the abstract model

- add the operation to the concrete model; typecheck
- modify any global property proofs about the concrete model
- add a new branch to the refinement proof to cover the new operation

■

Add a Function Argument

Problem : A functionality change a new argument to a function.

Solution : First add the argument, then add the constraints separately.

■

Part III

Wrapping up

Chapter 18

Tool support

18.1 Introduction

The use of (generative sets of) patterns to produce Z specifications and to refactor existing specifications goes some way towards facilitating the use of Z – particularly in conjunction with existing Z tools.

However, commercial specification developers need more, and better-targetted, tools. The required tools should both exploit and support patterns and refactoring:

- Where a pattern or some part of it is fixed-form, a tool should support it directly (perhaps as a built-in generic instantiated by the user).
- Where a pattern provides a template, or where various forms apply in different contexts, the tool should guide the user.
- New Z tools should aim to support documentation formats, presentation patterns and alternation between well-defined choice patterns.
- Existing Z tools might be refactored to exploit similar patterns directly.

18.2 Existing Tool Support

Tool support for patterns is an important requirement for the (industrial) specification development process. Manual application of patterns during development is difficult – even simple things like name consistently can be overlooked, especially if the naming convention is evolving with the specification. Conforming to patterns during maintenance is even more difficult, especially if the particular patterns used have not been documented.

In the object-oriented community, there is work on integrating patterns into the development process. Some schemes have been invented for encoding patterns in classes, but the match is not good. Patterns are at a different level from the language constructs. Mostly, naming conventions and comments are used to indicate the presence or use of patterns in the code. But some tool support is possible, both in software development and in formal notations. Rather than try to invent a new general purpose meta-language to support the identified patterns, tool developers should first concentrate on supporting the individual patterns explicitly.

Current Z tools are mostly syntax- and type-checkers, and proof tools. Compared to programming language IDEs, they provide relatively unsophisticated development environments.

The presentation patterns are supported to a greater or lesser extent by current Z tools, but even there little is automatic. Existing tools format to expose structure: most LaTeX-based tools, for example, [Spivey 1992a], give the user complete control over line breaks, indentation and white space within phrases; other tools such as CADiZ [Toyn 2001] and Formaliser [Stepney 2001] have automatic 'pretty-printing' layouts, but they do not always give optimal readability, and are not configurable to different layout standards. Similarly, provide navigation is supported in LaTeX-based tools via the LaTeX `\index` command and in HTML-based tools via hyperlinks. Z-Eves [Saaltink 1997] has a good navigation interface. Name consistently is partially supported by search and replace; such operations are even more useful when scope-sensitive. Graphical tools for GUI design show how a tool can automatically generate underlying code such that user changes to that code are reflected back in the GUI design. A similar approach could be used to support many of the Z patterns.

Z development process patterns are, surprisingly, the best supported, because many of the identified patterns are validation patterns requiring proof, for which proof tools exist. But even proof tools provide little *explicit* support for validation proof patterns – they are general purpose, rather than sensitive to the particular proofs required. There is some support for animation, ranging from conventions for semi-automatic translation of Z to an executable form [West & Eaglestone 1992], [Hewitt *et al.* 1997], to executable subsets of Z itself [Valentine 1992]. Making such conventions and tools sensitive to particular structural patterns would greatly smooth the process.

Z tools implement various patterns and refactorings. Tools such as CADiZ provide sanity checks such as precondition calculation, *schema expansion*, and other rewritings that preserve the mathematical semantics of the Z specification. The

current motivation for these refactorings is the need to provide normalised specification components for mechanised proof assistance. The profile of the software engineering motivations needs to be raised!

18.3 A better way of supporting patterns

Template support for presentation patterns such as comment the intention could be provided: addition of a declaration or predicate would cause a new comment line to be provided, with a prompt to the developer to explain the addition (either on the comment line or in an existing comment line). The tool should manage the linking of comment lines to the Z lines; reordering of declaration or predicates should cause corresponding reordering of their associated comments.

Tool support for choice patterns would allow users to change their minds. As a trivial example, Formaliser can convert between a horizontal and vertical schema display. Support for modelling product types, to assist in changing the representation between schemas and Cartesian products, could be implemented by adaptation from existing tool support for *schema expansion*.

Architecture patterns are not yet well supported by tools. Delta/Xi is supported, simply because its naming conventions are partly encoded in the Z core language, yet tools need to 'understand' whether a particular schema is a state, an operation, or a piece of scaffolding.

Interactive support in the form of state and operation templates, framing schema templates for promotion, and function definitions broken down over free types for morph, could all be automated. The Delta/Xi diagrams presented above show the schema components and their interrelationships; these could form a framework for "intelligent" architectural support, extending the existing tool facilities for tracking named component usage in a specification.

One day, tools may be configurable to support different specification aims (readability, provability, etc). They may be able to detect antipatterns in each style of specification, and able to guide the developer to a better representation, based on the patterns appropriate to that form of specification.

18.4 Tool support for refactoring

A code refactoring can affect an entire program. For example, changing the name of a method involves a change at every place that method is called. However, although widespread, such a change is shallow. Deeper code refactorings tend to be confined to small regions of code. It is by having *small* changes (either small in depth, or small in breadth) that code refactoring is manageable.

Formal specification and proof refactorings tend to have more widespread effects, because of the impact of a specification change on the proofs and later development steps. For example, adding a component or predicate to the state may involve adding it to the proof of every operation. Again, the changes tend to be shallow: the addition will probably have little or no effect on most operations. However, every proof needs to be checked, and this can be tedious without suitable tool support.

Tools are currently being built to support code refactorings. For example, the latest version of JBuilder includes a refactoring wizard; the Demeter tool (Northeastern University, Boston, US) supports generic Java programming, and a range of refactoring and reuse facilities [Demeter Research Group 1999]. Similar tool support (within a proof tool) for formal refactoring would be a benefit. Such a tool could support common refactorings such as those described earlier

Also important is support for pushing a relatively small change through the proof. This requires facilities for parameterising tactics for machine-generated proofs. This is a matter of on-going research.

Chapter 19

Conclusion

Z patterns have something in common with many of the software engineering pattern languages.

- [Beck 1997]'s Smalltalk coding standards provide inspiration for commenting and presentations
- [Gamma *et al.* 1995]'s design patterns are similar in intent to many of the presentation and structural Z patterns
- [Larman 2001]'s UML patterns can be compared to the higher-level architecture patterns, and the generative use of patterns

A Pattern Language for Z, which is essentially a packaging of existing language elements and usage according to their context of use, will help to make explicit the wider range of conventions and styles available. In addition, it will help to provide good solutions to well-known recurrent problems.

Refactoring is a natural partner of patterns. The patterns provide refactoring targets, and alert developers to potential refactorings. Everyone refactors, as understanding of the system being described or experience of the description language evolves. The trick is to do refactoring in a controlled, reversible manner. This requires tool support, for the modelling language, and for the development process which is the context for the use of the language. It also requires a good understanding of,

- the purpose of the model (communication, proof, or whatever)
- the required form of meaning preservation, so that the legitimacy of the refactoring can be demonstrated.

Appendix A

Diagram illustrations

A.1 Delta/Xi pattern diagrams

This section illustrates the use of Delta/Xi : diagram the structure on a large real-world specification, that of the electronic Purse in [Stepney *et al.* 2000].

It makes use of several features:

- dotted boxes for items not occurring explicitly in the specification
- indicating multiple similar schemas with overlapping boxes (and a naming convention)
- splitting the diagram into three, and using rounded boxes to indicate overlap
- highlighting the use of the promotion pattern (shaded boxes)

A.2 Morph pattern diagrams

This section illustrates the use of Morph : diagram the structure on a medium-sized example specification, that of the toy compiler in [Stepney 1993], and on a large real-world specification, that of the real compiler in [Stepney & Nabney 2003].

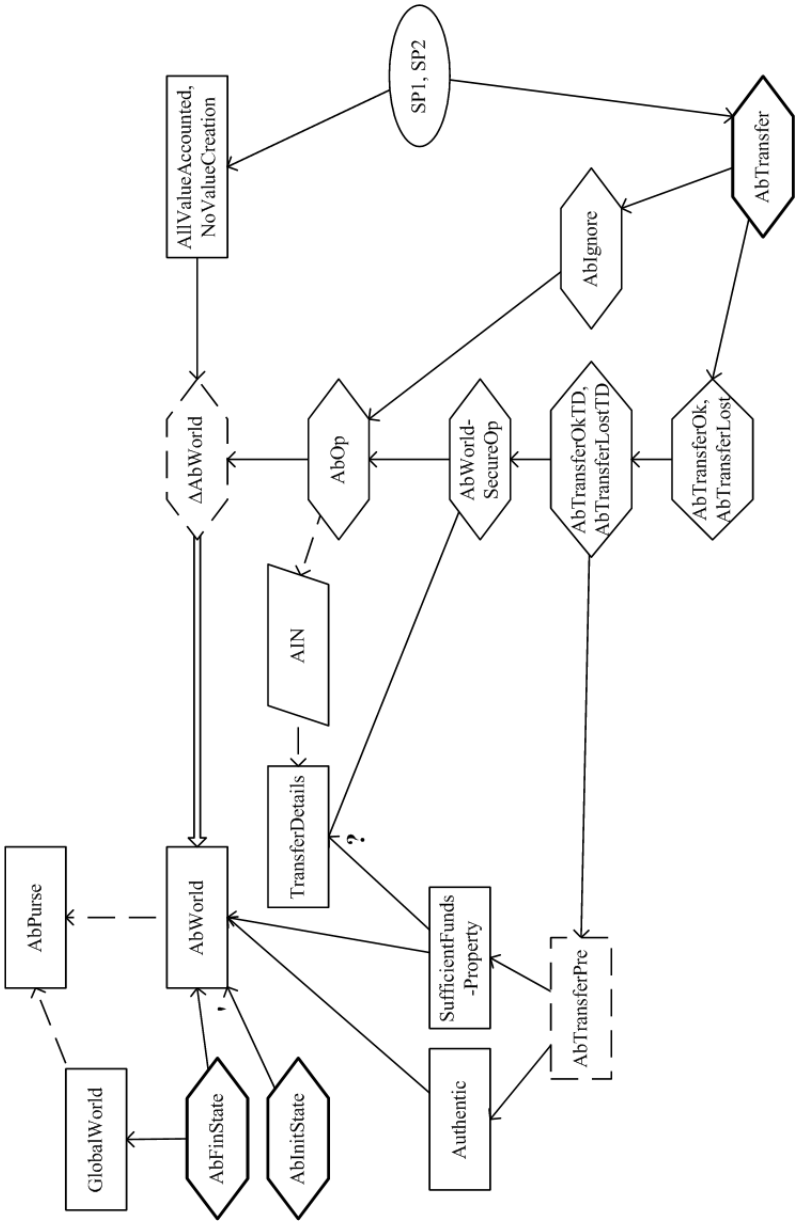

Figure A.1 Diagram of the structure of the abstract Purse model [Stepney *et al.* 2000, chapter 3].

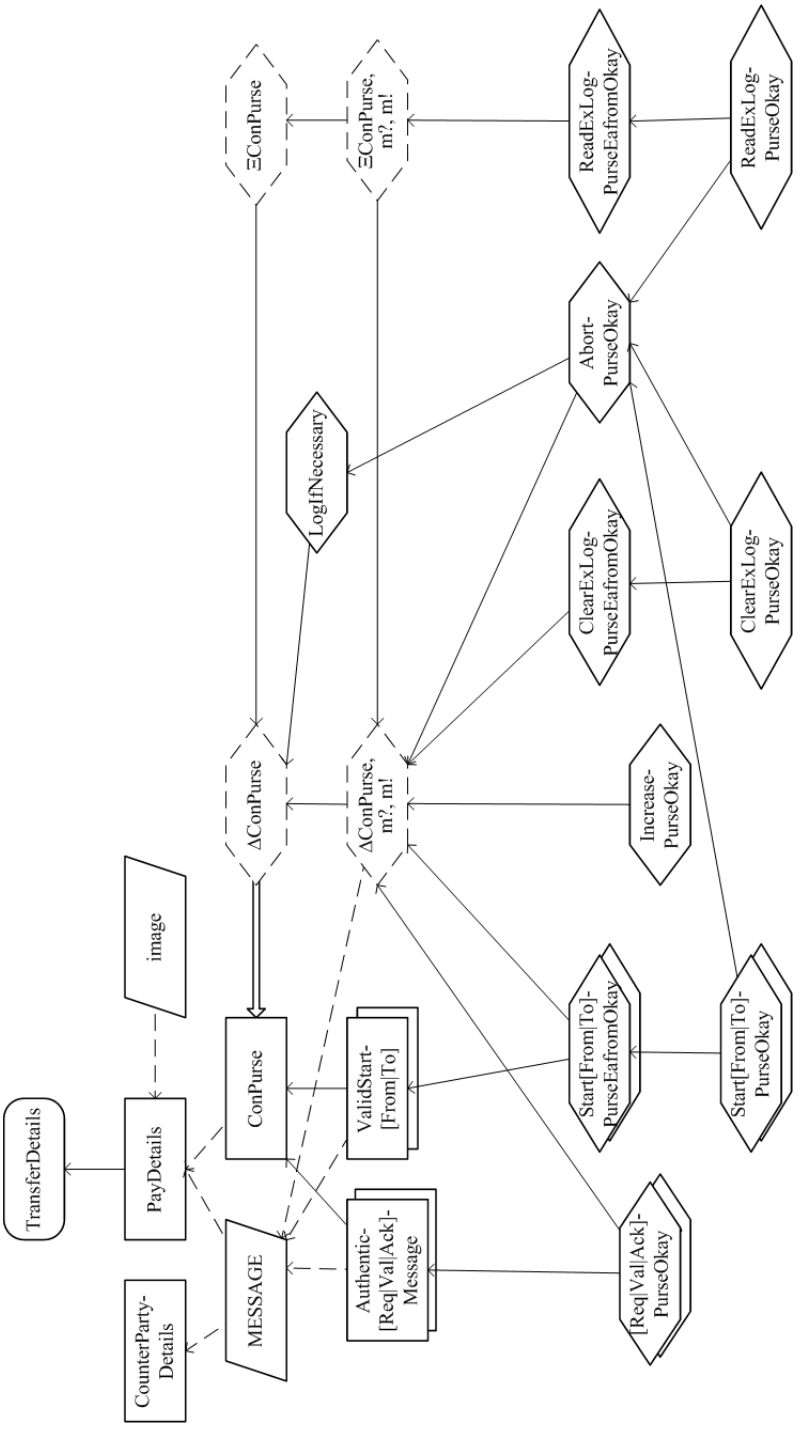

Figure A.2 Diagram of the structure of the concrete Purse model, local state [Stepney *et al.* 2000, chapter 4].

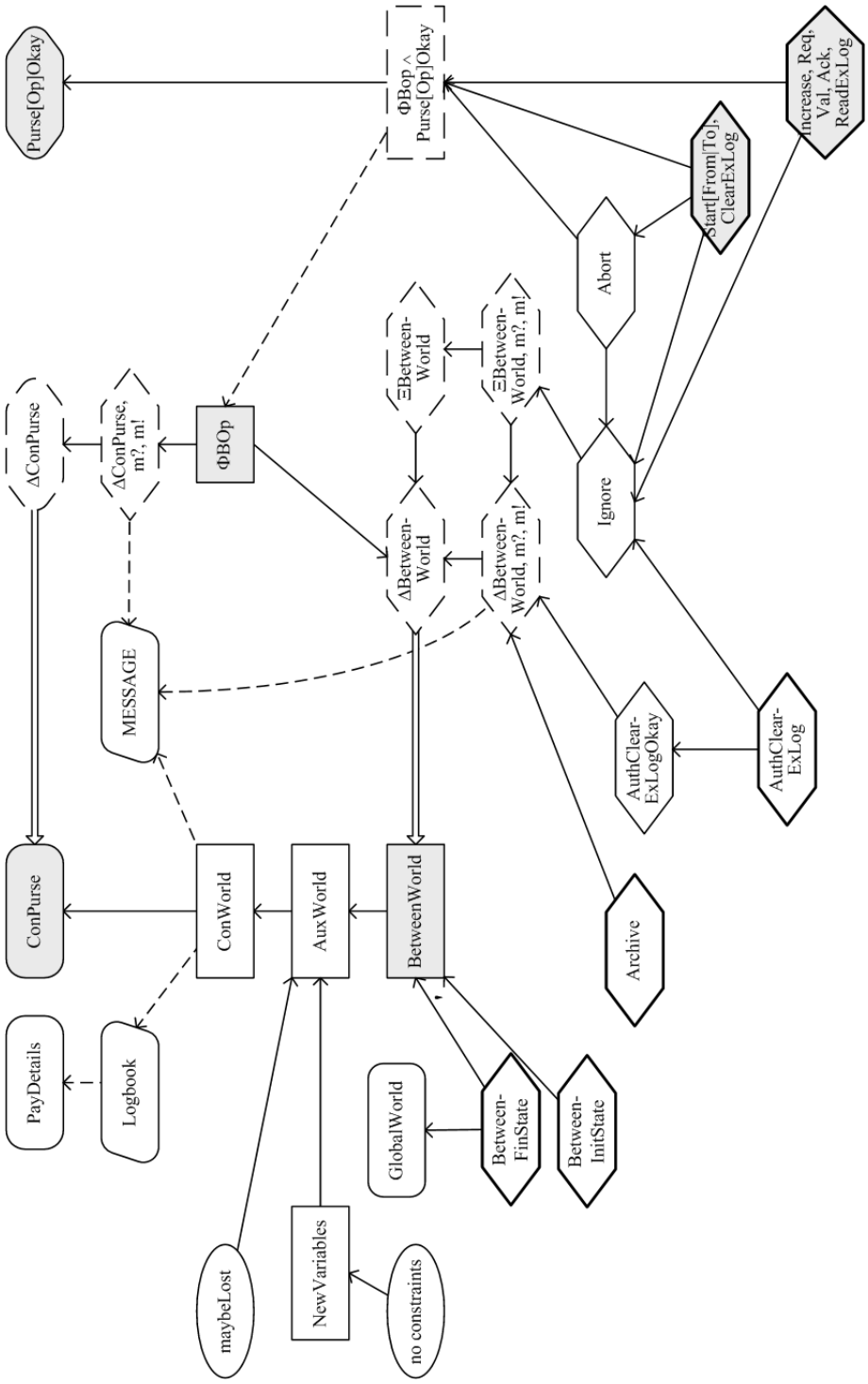

Figure A.3 Diagram of the structure of the concrete Purse model, global state [Stepney *et al.* 2000, chapter 5], promotion pattern highlighted.

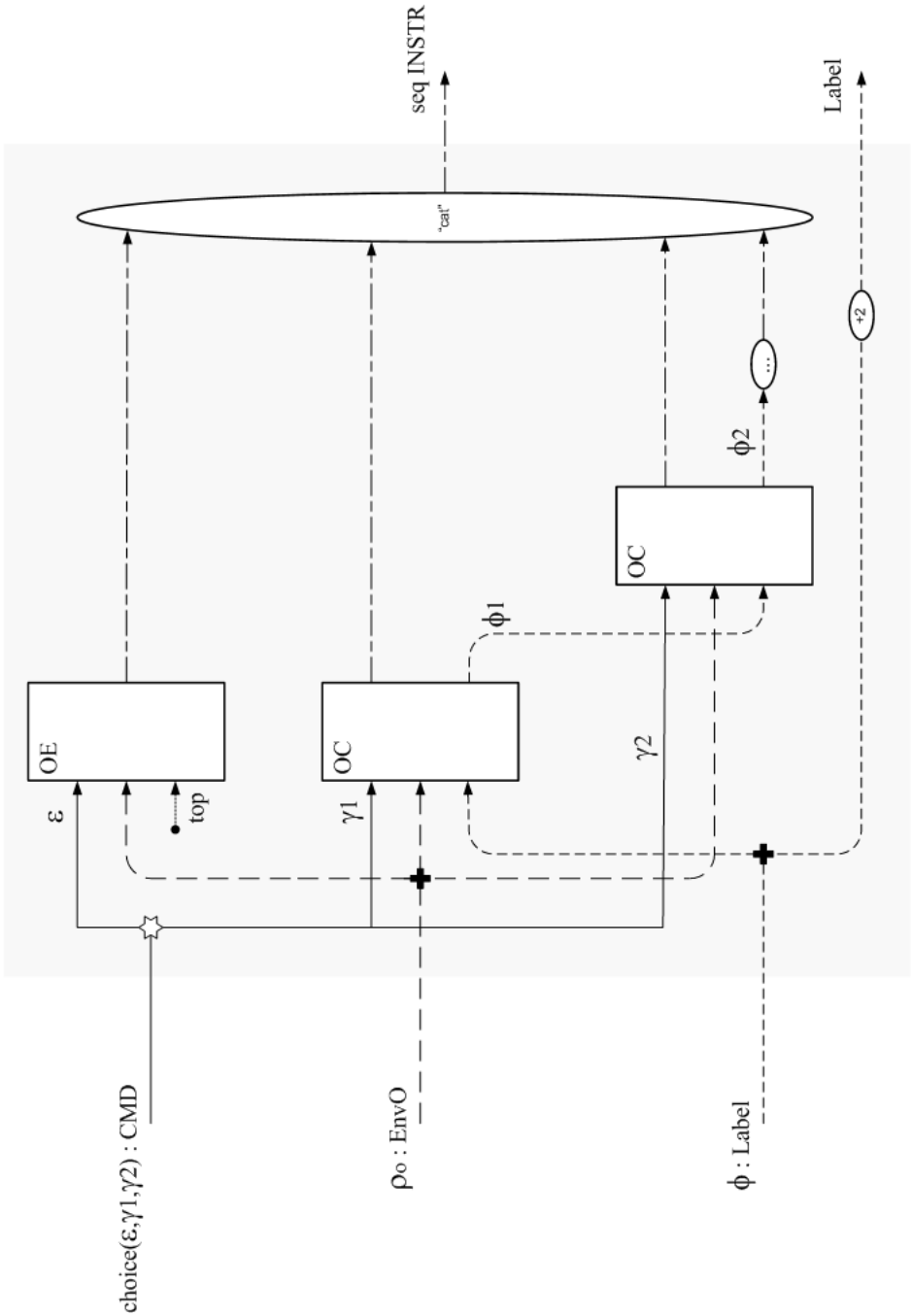

Figure A.4 Diagram of the structure of the Tosca command compile function, applied to the choice command [Stepney 1993, section 11.4.5].

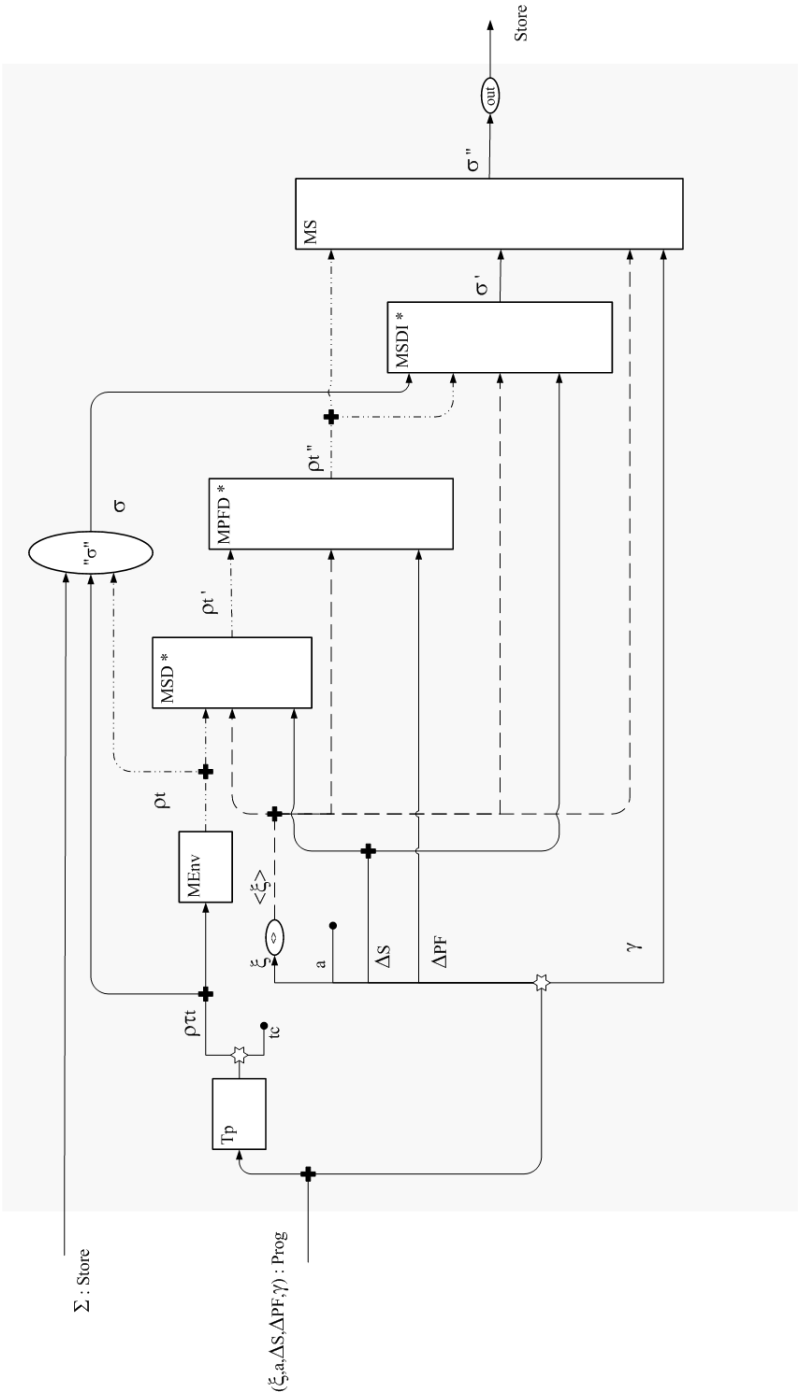

Figure A.5 Diagram of the structure of the DeCCo Pasp program meaning function [Stepney & Nabney 2003, section I.14.4].

Appendix B

Bibliography

[Alexander *et al.* 1977]
Christopher Alexander, Sara Ishikawa, Murray Silverstein, Max Jacobson, Ingrid Fiksdahl-King, and Shlomo Angel. *A Pattern Language: Towns, Buildings, Construction*. Oxford University Press, 1977.

[Arthan 2000]
Rob Arthan. Analysis of compiled code: a prototype formal model. In [Bowen *et al.* 2000], pages 433–449.

[Barden *et al.* 1994]
Rosalind Barden, Susan Stepney, and David Cooper. *Z in Practice*. BCS Practitioners Series. Prentice Hall, 1994.

[Basin *et al.* 2002]
D. Basin, R. Rittinger, and L. Viganò. Analysis of the CORBA security service. In [Bert *et al.* 2002].

[Beck 1997]
Kent Beck. *Smalltalk Best Practice Patterns*. Prentice Hall, 1997.

[Bert *et al.* 2002]
Didier Bert, Jonathan P. Bowen, Martin C. Henson, and Ken Robinson, editors. *ZB2002: Second International Conference of B and Z Users, Grenoble, January 2002*, volume 2272 of *LNCS*. Springer Verlag, 2002.

[Bowen *et al.* 2000]
Jonathan P. Bowen, Steve Dunne, Andy Galloway, and Steve King, editors. *ZB2000: First International Conference of B and Z Users, York, August 2000*, volume 1878 of *LNCS*. Springer Verlag, 2000.

[Brown *et al.* 1998]

William J. Brown, Raphael C. Malveau, Hays W. McCormick III, and Thomas J. Mowbray. *AntiPatterns*. Wiley, 1998.

[Brown 1979]
Peter J. Brown. *Writing Interactive Compilers and Interpreters*. Wiley, 1979.

[Cooper & Stepney 2000]
David Cooper and Susan Stepney. Segregation with communication. In [Bowen *et al.* 2000], pages 451–470.

[Coplien 1995]
James O. Coplien. A generative development-process pattern language. In James O. Coplien and Douglas C. Schmidt, editors, *Pattern Languages of Program Design*. Addison-Wesley, 1995.

[Demeter Research Group 1999]
Demeter Research Group. DJ web pages.
http://www.ccs.neu.edu/research/demeter/DJ/DJ-product-guide.html, 1999.

[d'Inverno & Luck 2001]
Mark d'Inverno and Michael Luck. *Understanding Agent Systems*. Springer Verlag, 2001.

[d'Inverno & Priestly 1995]
Mark d'Inverno and M. Priestly. Structuring specification in Z to build a unifying framework for hypertext systems. In *Ninth International Conference of Z Users, Limerick, Ireland, September 1995*, volume 967 of *LNCS*, pages 83–102. Springer Verlag, 1995.

[Duke & Rose 2000]
Roger Duke and Gordon Rose. *Formal Object-Oriented Specification Using Object-Z*. Macmillan, 2000.

[Dyer 1992]
M. Dyer. *The Cleanroom Approach to Quality Software Development*. Wiley, 1992.

[Easterbrook *et al.* 1996]
S. Easterbrook, R. Lutz, J. Covington, J. Kelly, M. Ampo, and D. Hamilton. Experiences using formal methods for requirement modeling. NASA/WVU Software Research Lab, Fairmont, WV, Technical Report NASA-IVV-96-018, 1996.

[Flinn & Sørensen 1992]
Bill Flinn and Ib Holm Sørensen. Specification of the UNIX filing system. In [Hayes 1992].

[Fowler 1997]
Martin Fowler. *Analysis Patterns*. Addison-Wesley, 1997.

[Fowler 1999]
Martin Fowler. *Refactoring: improving the design of existing code*. Addison-Wesley, 1999.

[France 2001]
Robert France, 2001. private communication.

[Gamma *et al.* 1995]
Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns*. Addison-Wesley, 1995.

[Goguen & Malcolm 1996]
Joseph Goguen and Grant Malcolm. *Algebraic Semantics of Imperative Programs*. MIT Press, 1996.

[Hall 1990]
J. Anthony Hall. Using Z as a specification calculus for object-oriented systems. In Dines Bjorner, C. A. R. Hoare, and H. Langmaack, editors, *VDM'90: VDM and Z – Formal Methods in Software Development, Kiel*, volume 428 of *LNCS*, pages 290–318. Springer Verlag, 1990.

[Hayes 1992]
Ian Hayes, editor. *Specification Case Studies*. Prentice Hall, second edition, 1992.

[Heisel & Souquiéres 1999]
Maritta Heisel and Jeanine Souquiéres. A method for requirements elicitation and formal specification. Industrial tutorial presented at *FM'99: World Congress on Formal Methods*, Toulouse, 1999.

[Hewitt *et al.* 1997]
M. A. Hewitt, C. M. O'Halloran, and C. T. Sennett. Experiences with PiZA, an animator for Z. In J. P. Bowen, M. G. Hinchey, and D. Till, editors, *ZUM'97: The Z Formal Specification Notation*, volume 1212 of *LNCS*, pages 37–51. Springer-Verlag, 1997.

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

[ISO-Z 2002]
ISO/IEC 13568. *Information Technology—Z Formal Specification Notation—Syntax, Type System and Semantics: International Standard*, 2002.

[Larman 2001]
Craig Larman. *Applying UML and Patterns, 2nd edition*. Prentice Hall, 2001.

[Lopes & Lieberherr 1994]
Cristina Videira Lopes and Karl Lieberherr. Generative patterns. In *ECOOP'94 Workshop on Patterns, Bologna, Italy*, 1994.

[Mander & Polack 1995]
Keith C. Mander and Fiona Polack. Rigorous specification using structured systems analysis and Z. *Information and Software Technology*, 37(5):285–291, 1995.

[Matthews & Swatman 2000]
Chris Matthews and Paul A. Swatman. Fuzzy concepts and formal methods: A fuzzy logic toolkit for Z. In [Bowen *et al.* 2000], pages 491–510.

[Morgan & Suffrin 1992]
Carroll Morgan and Bernard Suffrin. Specification of the UNIX filing system. In [Hayes 1992].

[Polack & Stepney 1999]
Fiona Polack and Susan Stepney. System development using Z generics. In *FM99, Toulouse, France*, volume 1708 and 1709 of *LNCS*, pages 1048–1067. Springer Verlag, 1999.

[Polack *et al.* 1993]
Fiona Polack, Mark Whiston, and Keith C. Mander. The SAZ project : Integrating SSADM and Z. In *FME'93 : Industrial Strength Formal Methods, Odense, Denmark*, volume 670 of *LNCS*, pages 541–557. Springer Verlag, April 1993.

[Saaltink 1997]
Mark Saaltink. The Z/EVES system. In *ZUM'97: The Z Formal Specification Notation*, LNCS 1212. Springer, 1997.

[Semmens *et al.* 1992]
L. T. Semmens, R. B. France, and T. W. Docker. Integrated structured analysis and formal specification techniques. *The Computer Journal*, 35(6), 1992.

[Smith 2000]
Graeme Smith. *The Object-Z Specification Language*. Kluwer, 2000.

[Spivey 1992a]
J. Michael Spivey. *The fuzz Manual*. The Spivey Partnership, 2nd edition, 1992. ftp://ftp.comlab.ox.ac.uk/pub/Zforum/fuzz.

[Spivey 1992b]
J. Michael Spivey. *The Z Notation: a Reference Manual*. Prentice Hall, 2nd edition, 1992.

[Stepney & Cooper 2000]
Susan Stepney and David Cooper. Formal methods for industrial products. In [Bowen *et al.* 2000], pages 374–393.

[Stepney & Cooper 2003]
Susan Stepney and David Cooper. Smart card operating system: Specification, refinement, and proof. Technical Report YCS-2002-???, York, 2003.

[Stepney & Nabney 2003]
Susan Stepney and Ian Nabney. The DeCCo papers. Technical Report YCS-2003-???, York, 2003.

[Stepney *et al.* 1992]
Susan Stepney, Rosalind Barden, and David Cooper, editors. *Object Orientation in Z*. Springer Verlag, 1992.

[Stepney *et al.* 2000]
Susan Stepney, David Cooper, and Jim Woodcock. An electronic purse: Specification, refinement, and proof. Technical Monograph PRG-126, Programming Research Group, Oxford University Computing Laboratory, 2000.

[Stepney 1993]
Susan Stepney. *High Integrity Compilation: A Case Study*. Prentice Hall, 1993.

[Stepney 1998]
Susan Stepney. A tale of two proofs. In *BCS-FACS Third Northern Formal Methods Workshop*, 1998.

[Stepney 2001]
Susan Stepney. Formaliser Home Page.
http://public.logica.com/~formaliser/, 2001.

[Toyn 2001]
Ian Toyn. CADiZ web pages.
http://www-users.cs.york.ac.uk/~ian/cadiz/, 2001.

[Valentine *et al.* 2000]
Samuel H. Valentine, Ian Toyn, Susan Stepney, and Steve King. Type-constrained generics for Z. In [Bowen *et al.* 2000], pages 250–263.

[Valentine 1992]
Samuel H. Valentine. Z^{--}, an executable subset of Z. In J. E. Nicholls, editor, *Z User Workshop, York 1991*, Workshops in Computing, pages 157–187. Springer-Verlag, 1992.

[Valentine 1993]
Samuel H. Valentine. Putting numbers into the mathematical toolkit. In Jonathan P. Bowen and John E. Nicholls, editors, *Proceedings of the 7th Annual Z User Meeting, London 1992*, Workshops in Computing. Springer Verlag, 1993.

[West & Eaglestone 1992]
M. M. West and B. M. Eaglestone. Software development: Two approaches to animation of Z specifications using Prolog. *IEE/BCS Software Engineering Journal*, 7(4):264–276, July 1992.

[Woodcock & Cavalcanti 2002]
J. C. P. Woodcock and A. L. C. Cavalcanti. The semantics of circus. In [Bert *et al.* 2002].

[Woodcock & Davies 1996]
J. C. P. Woodcock and J. Davies. *Using Z. Specification, Refinement, and Proof*. Prentice–Hall, 1996.

[Woodcock & Loomes 1988]
Jim Woodcock and Martin Loomes. *Software Engineering Mathematics*. Pitman, 1988.

[Wordsworth 1987]
J. B. Wordsworth. Formal methods in the development of CICS. *BCS Computer Bulletin*, December 1987.