An Outline Pattern Language for Z: five illustrations and two tables

Susan Stepney, Fiona Polack, and Ian Toyn

Department of Computer Science, University of York,
Heslington, York, YO10 5DD, UK.
{susan,fiona,ian}@cs.york.ac.uk

Abstract. We introduce a pattern language for using formal methods in computer system engineering. We focus on the Z notation, but many of the patterns are adaptable to other formal notations, or can be used to help choose a notation, or to decide on a style of formality. As in other pattern languages, we are providing a new presentation of existing practice, to make it accessible to computer systems engineering. We propose an initial classification of Z patterns, present selected examples, and outline issues of tool support.

Keyword: Z, patterns, development methods

1 Introduction

Formal methods have been used in computer systems development for decades. The most mature forms, particularly those 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, and are poorly supported by development tools.

In mature development approaches, the stages and steps of development are clear and generally-accepted, whereas 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 currently require specialist knowledge, in an area that is not generally recognised as meriting any development specialism.

2 Motivation

This paper is a contribution to the commercial acceptance of formality, specifically of Z. Z 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 a desire to make Z more usable by commercial non-specialist developers. Our reason for investigating patterns comes from experience in the industrial use of Z. One of the authors was a member of Logica UK's Formal Methods Team, 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 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.

Z textbooks introduce the mathematical bases 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 work in this paper is a new presentation of well-known material, with the concept of *pattern* applied to enhance the "semantic structure" of Z, thereby helping the writing, reading and presentation of Z. Formalisms such as Z have a role to play in general software development. So the patterns 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

3 Patterns

Patterns, originally introduced by [Alexander *et al.* 1977] in the context of architecture, 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].

Patterns do not stand in isolation. As [Alexander *et al.* 1977] explain, 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.

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. 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. A good pattern catalogue can be applied to assist all elements of construction of a description (program, design, etc).

Some patterns are general purpose, occurring in similar forms across many media (for example, across languages, development phases, contexts). For example, all notations require commentary which is clear, consistent, and adds meaning to the text, and all notations have common usage conventions that can be expressed as patterns.

Some patterns are specific to the language for which they are written. For example, [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 is using an architectural pattern other than Delta/Xi, then most of the patterns written for use with the Delta/Xi pattern (promotion, change part of the state etc) are irrelevant.

4 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 results.

In [Brown *et al.* 1998], most of the antipatterns describe universally poor practice. However, in other contexts, and particularly in notations such as Z, one developer's antipattern may be 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 communication; we are also working on patterns for other purposes including refinement, implementation and proof.

5 Patterns in Z

5.1 Motivation

Z provides a core language. (We refer to the two main variants of Z as ZRM [Spivey 1992b] and as ISO-Z [ISO-Z 2002]. By *Mathematical Toolkit*, we mean those well-known definitions in [Spivey 1992b, chapter 4] and [ISO-Z 2002, annex B].) 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).

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.

It is common for Z specifications to be coerced into these conventions, no matter how inappropriate. By separating out and describing toolkit patterns, and by naming the Delta/Xi pattern and its associated subpatterns, we hope to make it clear that these are just one choice of many.

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

5.2 Structure of 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 (see [Stepney *et al.* 2003b] for elaborations of the promotion pattern).

5.3 Diagram patterns

There are many diagram styles appropriate for summarising different Z structures. For example, Venn diagrams can be used to represent set-theoretic statements; state machines summarise event-based structures; data-flow diagrams can represent functional styles.

We have distilled diagrammatic sub-patterns for specifications using the Delta/Xi and morph architecture patterns. The full details of these can be found in [Stepney *et al.* 2003c] [Stepney *et al.* 2003a]; the Delta/Xi sub-pattern is outlined below.

6 Catalogue of Z patterns

6.1 Introduction

In writing about patterns, we use the following categories (further work is needed on the developing and refining these 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*.

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.

Table 1 categorises the Z patterns that we have identified so far.

6.2 Presentation patterns

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. These patterns 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]).

6.3 Idiom patterns

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.

	Sub-category		
Category	**Documentation**	**Style**	**Usage**
Presentation	Comment the intention	Format to expose structure	
	Provide navigation		
	Name consistently		
	Overlong name		
	Overmeaningful name		
Idiom		Assemble from chunks	
		Representing many-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	Use generics to control detail	Modelling optional elements
	Name predicates	*Fortran*	Modelling product types
		Sørensen shorty	Modelling membership or flags
			Boolean flag
			Partial precondition
Architecture		Morph	`Promotion`
		Event traces	`Delta/Xi`
		Object Orientation	
		Algebraic style	
		Goldilocks chunks	
		Unsuitable Delta/Xi pattern	
Domain		Application oriented theory	Domain specific toolkits
			Schema operator toolkit
Development		Focus the formality	`Do a refinement`
		Use integrated methods	Animate
			Do sanity checks
			Express implicit properties
			Apply syntax and type checking
			Prove rigorously
			Prove formally

Table 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.

6.4 Structure patterns

At the intermediate structural level, structure 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.

6.5 Architecture patterns

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 on to the solution architecture. We have identified some alternatives to the widely-used Delta/Xi pattern.

6.6 Domain patterns

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 provides 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 that further toolkit libraries will be developed with their associated patterns in future engineering support for Z-based formal methods.

Toolkits could range in level and sophistication, from straightforward extensions to the Mathematical Toolkit, to complete application-specific support with template structures, special operators, and a full set of laws.

6.7 Development patterns

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 notation. 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.

Do a refinement, a generative pattern with elaborations, is outlined briefly and illustrated using the diagram the structure pattern.

7 Five illustrative patterns

Table 1 shows the Z patterns that we have identified so far. We illustrate these with the examples below; they have been selected to illustrate the pattern structure, antipatterns, good Z style, and some of the new patterns available with ISO-Z. The promotion pattern, which is itself an elaboration of Delta/Xi, is covered in detail in [Stepney *et al.* 2003b]. The remaining patterns are described in the full Z pattern catalogue [Stepney *et al.* 2003c].

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:

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{Memory}} \\
\hline
ram, rom : ADDR \mathrel{\mkern6mu\mapstochar\mkern-6mu\rightarrow} BYTE \\
inmap, outmap : \mathbb{P}\, ADDR \\
\hline
\mathrm{disjoint}\langle \mathrm{dom}\, ram, \mathrm{dom}\, rom \rangle \\
inmap \cup outmap \subseteq \mathrm{dom}\, ram \\
\mathrm{disjoint}\langle inmap, outmap \rangle \\
\hline
\end{array}
$$

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.
– Where a Z paragraph has internal structure let the comment structure clearly follow that of the Z paragraph. For a schema, for example, precede the Z paragraph with commentary on declarations (global names), follow the paragraph with commentary on internals.

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

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{Memory}} \\
\hline
ram, rom : ADDR \mathrel{\mkern6mu\mapstochar\mkern-6mu\rightarrow} BYTE \\
inmap, outmap : \mathbb{P}\, ADDR \\
\hline
\mathrm{disjoint}\langle \mathrm{dom}\, ram, \mathrm{dom}\, rom \rangle \\
inmap \cup outmap \subseteq \mathrm{dom}\, ram \\
\mathrm{disjoint}\langle inmap, outmap \rangle \\
\hline
\end{array}
$$

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

□

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".
□

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 order 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, as in $\mu S \bullet foo = x \land bar = y$); 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, as in $\langle\!| foo == x, bar == y |\!\rangle$, and tuple components to be directly selected, as in $t.3$, but there is still a difference in the terseness of construction.
□

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.)
- for schema inclusion, use solid arrows pointing from the including schema to 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 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.

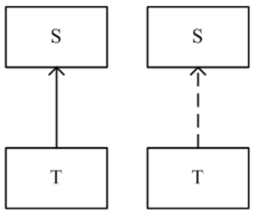

Fig. 1. (a) schema T includes schema S. (b) schema T references schema S.

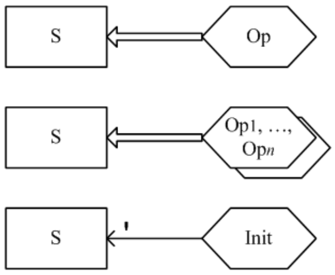

Fig. 2. (a) operation *Op* includes ΔS. (b) several operations Op_i include ΔS. (c) initialisation operation *Init* includes $S\,'$.

- If schema T includes schema S, either as declaration as $T == [\ S;\ \ldots \mid \ldots\ ]$, or as a predicate as $T == [\ \ldots \mid S \wedge \ldots\ ]$, it can be drawn as figure 1a.
- If schema T refers to schema S other than by inclusion, for example as $T == [\ f : x \twoheadrightarrow S \ldots \mid \ldots\ ]$, it can be drawn as figure 1(b)
- If operation schema *Op* includes schema S as $Op == [\ \Delta S \ldots \mid \ldots\ ]$, it can be drawn as figure 2(a)
- 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 with Op_i in figure 2(b)
- If initialisation schema *Init* includes schema S as $Init == [\ S\,' \ldots \mid \ldots\ ]$, it can be drawn as figure 2(c)

Illustration: [Stepney *et al.* 2003c] shows the diagrams of a large Delta/Xi specification.
Variants:

- The notation can be extended to show conjectures (as in the do a refinement pattern, below).
  - conjectures are drawn in an oval, labelled with a suitable name, pointing to referenced schemas
- 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.

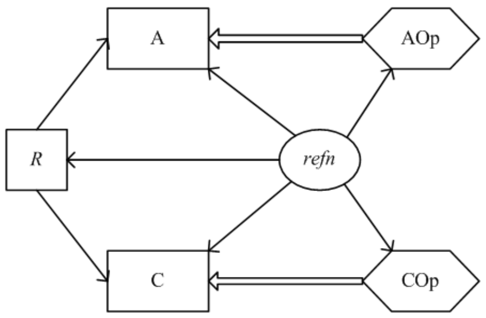

Fig. 3. Structure of the do a refinement pattern using diagram the structure.

- schemas or other data type boxes occurring in more than one sub-diagram are represented as rounded boxes on subsequent occurrences (see the example in [Stepney *et al.* 2003c]).

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

Specimen: this pattern is adapted from [d'Inverno & Luck 2001].
□

Do a refinement

Intent: connect, by formal proof of a refinement conjecture, specifications of the same system at different levels of abstraction.
Solution: Use the generative refinement patterns:

– Do a refinement: abstract model – provide an abstract specification
– Do a refinement: concrete model – provide an equivalent specification at a lower level of detail
– Do a refinement: retrieve relation – formally express the mapping between each abstract component and its concrete equivalent
– Do a refinement: conjecture – prove that the concrete operations and invariant maintain the invariant of the abstract specification

The underlying structure of a refinement is summarised in figure 3. The retrieve relation is the schema R. The refinement conjecture is represented as the ellipse, *refn*.

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 4.

The do a refinement pattern has a large number of elaborations, some of which are listed in table 2, below.
□

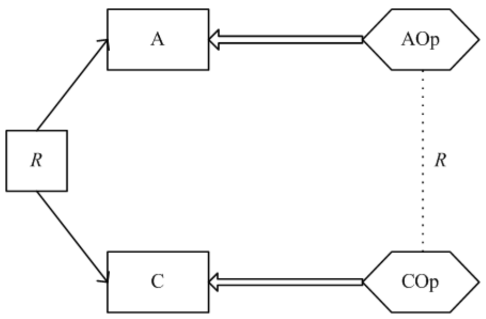

Fig. 4. Structure of the do a refinement pattern, abbreviated form.

8 Z 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 specific elements of a program (classes, structures, generic operations etc).

It is impossible to automatically generate a specification from a meta-language template alone. 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.

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. They form the beginnings of a method for Z. 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 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 semantics, 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 abstraction).

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 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 architecture 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.
`Do a refinement:` abstract model concrete model retrieve relation refinement 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	Apply reverse refinement rules
	blocking semantics	Allow a blocking semantics in concrete specification

Table 2. Generative Z Patterns and their Elaborations. Note that `promotion` is both an elaboration of `Delta/Xi`, and a generative parttern in its own right.

Table 2 identifies some generative patterns. The table is not exhaustive, either in terms of the list of generative patterns, or in the detail of elaborations and component patterns. See [Stepney *et al.* 2003b] for more details of the promotion generative patterns.

9 Tool support

9.1 Introduction

The use of (generative sets of) patterns to produce Z specifications goes some way towards a Z method – particularly in conjunction with existing Z tools. However, commercial specification developers need more, and better-targeted, tools. The required tools should both exploit and support patterns:

– Where a pattern or some part of it is fixed-form, a tool should support it directly (perhaps as a built-in generic pattern 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 patterns.

9.2 Existing Tool Support

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 use of patterns in the code. But some tool support is possible, both in software development and in formal notations. Before trying to invent a new general purpose meta-language to support the identified patterns, tool developers should 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 CADi$\mathbb{Z}$ [Toyn 2002] 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, perhaps 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 patterns would greatly smooth the process.

9.3 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 new 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" structural 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.

10 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 presentation. [Gamma *et al.* 1995]'s design patterns are similar in intent to many of the Z presentation, architecture and structure 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, helps to make explicit the wider range of conventions and styles available. In addition, it helps to provide good solutions to well-known recurrent problems.

References

[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.

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

[Bowen *et al.* 1997]
J. P. Bowen, M. G. Hinchey, and D. Till, editors. *ZUM'97: The Z Formal Specification Notation, 10th International Conference of Z Users, Reading, UK*, volume 1212 of *LNCS.* Springer, 1997.

[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.

[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.

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

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

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

[Hewitt *et al.* 1997]
M. A. Hewitt, C. M. O'Halloran, and C. T. Sennett. Experiences with PiZA, an animator for Z. In [Bowen *et al.* 1997], pages 37–51.

[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.

[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, 1993.

[Saaltink 1997]
Mark Saaltink. The Z/EVES system. In [Bowen *et al.* 1997], pages 72–85.

[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 2003]
Susan Stepney and David Cooper. Smart card operating system: Specification, refinement, and proof. Technical Report YCS-2003, York, 2003. (in press).

[Stepney & Nabney 2003]
Susan Stepney and Ian Nabney. The DeCCo papers. Technical Report YCS-2003, York, 2003. (in press).

[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 *et al.* 2003a]
Susan Stepney, Fiona Polack, and Ian Toyn. A meta-pattern for diagram patterns. 2003. (in preparation).

[Stepney *et al.* 2003b]
Susan Stepney, Fiona Polack, and Ian Toyn. Patterns to guide practical refactoring. 2003. (these proceedings).

[Stepney *et al.* 2003c]
Susan Stepney, Fiona Polack, and Ian Toyn. A Z patterns catalogue I: specification and refactoring, v0.1. Technical Report YCS-2003-349, York, 2003.

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

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

[Toyn 2002]
Ian Toyn. CADiZ web pages. http://www-users.cs.york.ac.uk/~ian/cadiz/, 2002.

[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, 1992.

[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.