Diagram Patterns and Meta-patterns to support formal modelling

with instantiations for B, CSP, Z, Circus, and CSP| |B

Susan Stepney, Fiona Polack, and Ian Toyn

University of York Technical Report YCS-2005-394

17 October 2005

Abstract

We present a pattern-based approach to depicting the structure of formal specifications. We propose a diagram meta-pattern, and instantiate it to produce a number of diagrammatic patterns for formal notations including B, CSP, and Z, and composite notations Circus and CSP| |B. The ultimate objective is to support practical use and reuse of specifications and their documentation.

Contents

Chapter 1

Introduction

This paper is part of ongoing research on making formal techniques of software development practically applicable. It brings together pattern concepts that address the need for systematic, repeatable approaches to such developments, with forms of illustration that can be used to summarise and explain formal text.

1.1 Diagrams are useful

We start from the observation that many people find diagrams useful to help them understand complex structures.

We introduce patterns that can be used to produce diagrams that are helpful *abstractions* of the formal text, and that can help to illustrate structure not immediately visible from the text. These are the kinds of diagrams people sketch for themselves when trying to understand a new concept in the formal text or formal model; in some cases, authors already provide these kinds of illustrations.

There are numerous kinds of model in a computer system development, representing different stages of development, different aspects of the system, different viewpoints, and different approaches to modelling. Diagrams can be used to summarise the structure of the formal model, to emphasise important parts of the model, whether structurally (the core or current focus of the formal text), or in terms of the domain modelled (for example, a controlling part of the system, the critical security or safety aspects of the system, or certain functionality).

In a diagram, it is easier than in formal text to apply typographic tricks to place or reduce emphasis, by, for example, use of scale, colour or shading – this is evident in computer-based presentations where current focus is highlighted visually using colour variations etc. Our diagrams are treated as part of the informal supporting commentary; they are *not* automatically generated from the formal text. (This should be contrasted with approaches where the diagrams themselves have 'rigorous' interpretations, for example, UML.) The author chooses which elements to emphasise, and which to elide, the better to explain the formal text to the reader.

Such diagrams can also contribute to the development process. An infelicity in the appearance of a diagram may indicate a corresponding infelicity in the structure of a model. If a diagram is produced after the model is completed (as is the case of all the examples in this paper), the result may indicate a *refactoring opportunity*: a chance to improve the model during subsequent development [Fowler 1999], [Stepney *et al.* 2002].

1.2 Patterns

The concept of patterns [Alexander *et al.* 1977] is increasingly used in computer systems analysis and development [Fowler 1997], [Gamma *et al.* 1995], [Coplien & Schmidt 1995], [Martin *et al.* 1998]. Patterns are characteristics of models that engineers wish to highlight. Sometimes a pattern is borrowed from another development (directly or via a pattern catalogue); sometimes it is simply a metaphor for a section of a one-off design – it may subsequently become a catalogued or reused pattern.

As with any work on patterns, this paper offers a new presentation of existing material. Existing concepts are applied to known formal idioms and structures, and general conclusions drawn that are of relevance to wider work on patterns and their representation, as well as for the practical use of formality.

Our experience of formal modelling indicates that patterns make formal modelling more practically attractive and help in the development of commercially-relevant support tools for formal specification, formal analysis, and formal development [Stepney *et al.* 2003a], [Stepney *et al.* 2003b], [Stepney *et al.* 2003c], [Valentine *et al.* 2004]. Diagrams are visual aids to representation or understanding of models. Our experience with software engineering patterns suggests that diagrammatic visualisations are a useful part of pattern descriptions, helping to record how the patterns are used. Our goal is, therefore, to identify diagrammatic representations for the identified formal patterns.

Our work on diagrams and diagram patterns for formal methods indicates that many of the desirable characteristics of the diagram patterns are also desirable

	specification	supporting documentation
formal	schema : Z scope	proof trees etc
informal		diagram patterns

Table 1.1 Examples of formal and informal diagrams used in the specification, or the supporting documentation

	Item	Relationship
UML class diagram	classes	associations, inheritance
UML sequence diagrams	objects, messages	temporal order
UML activity diagrams	activities	temporal ordering and flow
Cleanroom diagrams	functions	stimuli and responses
Jackson structures	states	decomposition, temporal order
Venn diagram	sets	union, intersection, etc
refinement square	states	operations, retrieve relation

Table 1.2 Examples of abstract items and relationships

characteristics in other areas of system modelling. The use of diagrams with formality must be undertaken in an informed way; this is equally true where concepts specified diagrammatically are re-engineered into formal descriptions. Table 1.1 summarises the potential interactions of formal text and diagrams.

We present many existing diagramming concepts and styles as patterns; we also introduce a few new contributions for the various formal notations.

1.3 Meta-concepts for diagrams

Many diagrams express two concepts of interest: the *items*, and the *relationships* between them. Table 1.2 gives abstract syntactic components of some common diagrams. In general, a suitable notation can be designed by deciding a graphical representation for the items and the relationships.

Table 1.3 gives some well-known concrete syntactic representations. We use this

Item	Relationship	Example
polygon	line, arrow	UML class diagram, Cleanroom diagram
boxes	lines, position	Jackson structures
polygon	polygon	Venn diagram
polygon	(position)	activity diagram
box	arrow	commuting square

Table 1.3 Examples of concrete representations of items and relationships

observation, that diagrams represent items and relationships using graphical symbols, to define a pattern for defining diagrams (section 1.4).

It is useful to remember that an abstract syntax can be represented by more than one concrete syntactic scheme. Equally, one concrete syntax can be used to represent different abstract concepts. This is both a strength and a weakness of structural diagrams. Metaphors can be introduced into new diagram styles with the deliberate intention of referencing another abstract syntax; however, metaphorical usage can also be assumed where none is intended. For example, data flow diagrams (in SSADM [CCTA 1990, Goodland & Slater 1995] or Yourdon [Yourdon 1989]) represent processes as boxes or ellipses, and data flow as arrows; many readers assume that the relative location of the processes on the page implies an ordering in their calling; in fact, there is no notational convention (in the cited methods) for identifying the triggering input(s) of a process or a sequence of processes. A pattern gives a concise summary of the author's intent for the notation, and may counter such mis-readings.

Observe that diagrams that try to express three concepts (perhaps, items plus static **and** temporal relationships) tend to be harder to read and construct than those which express only two concepts. For example, Jackson diagrams, used for entity life histories in methods such as SSADM, represent life-cycle components as boxes, organised into hierarchies using lines for static links. The life-cycle of an entity is described by the left-right position of the lowest leaves of every subtree in the hierarchy.

1.4 Meta-pattern ideas

In [Stepney *et al.* 2003c], [Stepney *et al.* 2003a] we introduce a terminology and notation for formal specification patterns. Textually, each pattern has a double bar header, several line-delimited sections, and ends with a □. Here we extend that notation to allow *meta-patterns*: patterns for writing patterns. The approach is similar to the general-purpose "pattern language for writing patterns" of [Meszaros & Doble 1998]. The meta-pattern is itself a pattern (distinguished from an ordinary pattern by its triple bar header), and its solution part includes an ordinary pattern template that can be instantiated to produce specific patterns.

Our meta-pattern for writing diagram patterns that capture the structural aspects of formal specifications is called Design the diagram.

Design the diagram

Intent

Give guidance for writing instantiated Diagram the structure patterns for particular formal modelling concepts.

Problem

It is difficult to visualise the structure of a large, complex specification. It can also be difficult to comprehend the details at a lower level. Diagrams help in many cases, but it is important that the diagram provides the right level of abstraction, and does not hinder understanding.

Solution

- Identify the Items and Relationships to be represented in a diagram.
- Identify how Items and Relationships are to be represented
- Instantiate the following template pattern, by providing values for the parts in angle brackets:

$\langle$X$\rangle$: Diagram the structure

Intent

Summarise [the ⟨A⟩ aspect of] the structure of a ⟨X⟩ specification using a diagram.

Problem

⟨An explanation of why that aspect of the structure benefits from being expressed diagrammatically.⟩

[**Example**]

⟨A typical case of a description (specification etc) that contains the structure but in which the structure is not as clear as it could be.⟩

Solution

- The items are ⟨ITEM⟩. Represent them as ⟨DIAGRAM ELEMENT (ITEM)⟩. [Label them with ⟨ITEM LABEL⟩.]
- The relationships are ⟨RELN⟩. Represent them as as ⟨DIAGRAM ELEMENT (RELN)⟩. [Label them with ⟨RELN LABEL⟩.]
- [Use ⟨EMPH⟩ to emphasise aspects of interest.]

Illustration

⟨Diagram the pattern structure⟩
⟨Diagram the example instance of the problem⟩

[**Constraint**]

⟨Give cases where the pattern is not useful.⟩

[**Variants**]

- When the specification contains ⟨VAR⟩, use ⟨DIAGRAM ELEMENT (VAR)⟩ to diagram it.
- Elide ⟨DIAGRAM ELEMENT⟩ under ⟨CIRCUMSTANCE⟩.

[**Related pattern**]

⟨If the recommended pattern is derived from an existing diagrammatic pattern, make the derivation clear: this should illuminate metaphorical aspects if these exist.⟩

[Specimens]

⟨Give examples in the literature where these, or similar, diagrams are used or defined.⟩

□

Illustration

The rest of this paper gives instantiations of this pattern.

Constraint

This pattern applies only to things that follow the ‘item, relationship’ metamodel. Don’t try to diagram too much detail.

Related pattern

The generated pattern should be a sub-pattern of a structural, or chunking, pattern in the relevant notation.

□

To illustrate instantiation of the meta-pattern, we choose examples drawn from the B, CSP and Z specification languages, and from the Circus (Z+CSP) and CSP| |B (B+CSP) combined languages.

Chapter 2

Application of Diagram Patterns to B

In the B method [Abrial 1996], [Schneider 2001], language and usage is tightly constrained to facilitate automated construction and discharge of proofs of consistency. In effect, the principal patterns of usage are built into the method by the support tools.

The aspect of B that can be most enhanced by diagrams is the structure of *machines* within a specification (or development – the B method supports specification, refinement and code generation). A number of diagrammatic styles appear in the literature. Laleau *et al.* [Laleau 2002], [Facon *et al.* 2000] use boxes to represent machines, and different styles of arrow to represent different machine structurings. An alternative diagrammatic approach, using nested ellipses and arrows, is used in [Polack 2003]. Here, we present a pattern based on diagrams in [Schneider 2001].

B : Diagram the structure

Intent

Summarise the machine structuring of a B specification using a diagram.

Problem

A B specification may comprise many machines, interlinked by B concepts such as **USES**, **INCLUDES**, **EXTENDS** and **PROMOTES**. The overall machine structure is hard to deduce from the B notation alone (or from machine dependency summaries produced by existing support tools).

Example

The following extends Schneider's *registrar* example [Schneider 2001, chapter 11]. The specification comprises B machines modelling aspects of a population registration system. We summarise the relevant B machines, giving only their structuring clauses, key state elements, and operation headers.

A machine, *Life*, defines the variables *man* and *woman*, and operations *born* and *die* that add and delete instances:

```
MACHINE Life
SETS PERSON;  SEX = {male, female}
VARIABLES man, woman
...
OPERATIONS
    born()
    die()
END
```

The machine *Marriage* needs access to the *Life* data structure but does not change its state; it has a **USES** clause referencing *Life*:

```
MACHINE Marriage
USES Life
VARIABLES marriage
...
OPERATIONS
    wed()
    part()
    partner()
END
```

The overall system machine, *Registrar*, specifies an interface to the operations of the *Life* and *Marriage* machines. The **EXTENDS** clause indicates that the *Marriage* machine is included, such that all its operations are promoted unaltered to the interface of *Registrar*. *Registrar* defines the effect on a marriage of the death of one of the partners in the externally-visible operation, *dies*. This calls the *die* operation of the *Life* machine. *Registrar* gains access to the operations of *Life* using an **INCLUDES** clause. The other operation of *Life*, namely *born*, is promoted to the interface in a **PROMOTES** clause.

MACHINE *Registrar*
EXTENDS *Marriage*
INCLUDES *Life*
PROMOTES *born*
OPERATIONS
  dies()
END

Finally, there is an independent *Dating* machine. This machine accesses data relating to unmarried individuals, and provides an operation, *match*. *Dating* has its own interface to the outside world:

MACHINE *Dating*
USES *Marriage*, *Life*
VARIABLES *boyfriend*, *girlfriend*
...
OPERATIONS
  match()
END

The example demonstrates the different ways by which machines are interlinked: **USES** for accessing but not changing state; **INCLUDES** for access to operations, with **PROMOTES** to indicate which operations are externally visible to the including machine; **EXTENDS** to encompass **INCLUDES** and **PROMOTES** when all operations of the included machine are to be promoted. The nesting and the external interfaces are difficult to visualise in the standard machine structures.

Solution

Use boxes to represent B machines. Represent interfaces among machines as follows:

- represent **INCLUDES**, and the including element of **EXTENDS**, by nesting the included machine inside the including machine
- indicate **USES** by an arrow from the using to the used machine

In addition, show the operations of each machine as tabs from the side of the machine in which they are defined, protruding as far as they are accessible. Thus, the operations of a machine that is referenced in the **EXTENDS** clause of an-

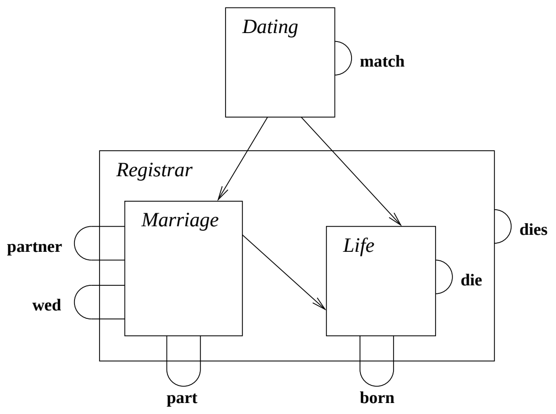

Figure 2.1 B diagram: the example's B machine structure (extended from [Schneider 2001, p171])

other machine protrude to the outside of the extending machine, as do operations referenced in a **PROMOTES** clause.

Label machines and operations with names from the B machines.

Use positioning to emphasise aspects of interest. For example, it may be helpful to direct **USES** arrows downwards from the using to the used machine.

Illustration

Figure 2.1 extends the diagram from Schneider's example; the *Dating* machine has been added to illustrate the **USES** clause in full.

Constraint

Because of the nesting of boxes, the diagrams would be inappropriate for high degrees of machine inclusion.

□

Chapter 3

Application of Diagram Patterns to CSP

Communicating Sequential Processes (CSP) [Hoare 1985], [Roscoe 1997], [Schneider 2000] is an algebra for expressing features of concurrent systems, including protocols and other communicating programs. The commercial potential of CSP is enhanced by Roscoe's model checker, FDR, which analyses the failure-divergence refinements of CSP specifications [FDR 2000], [Roscoe 1994].

CSP models processes and traces. To date, we have not found any useful diagrams for CSP trace models. This may be an example of where the textual form is more useful than a diagrammatic form. However, CSP process models do lend themselves to the use of diagrams (as, indeed, was recognised by CSP's inventor [Hoare 1985, section 1.2]). See also [Brooke & Paige 2002].

A CSP process specification has two main levels of structure.

- a single sequential process and its state transitions
- a collection of processes communicating by shared events on named channels

We define a pattern for each of these, and then illustrate their combination into the full CSP pattern.

3.1 Processes: state transitions

CSP Transitions: diagram the structure

Intent

Summarise the structure of a CSP process state transition specification using a diagram.

Problem

A CSP process specification contains a description of a process's internal behaviour. It can be difficult to determine the structure of a whole process from the structure of its parts.

Solution

The representation of states and state transitions is adapted from the Harel Statechart [Harel 1987] and UML state diagrams [Booch *et al.* 1999] notations.

- Represent states as rounded boxes, labelled with the state name.
- Represent transitions as arrows between the states, labelled with the events and guards. Indicate the initial state by an incoming arrow, if necessary. Label the arrow with an event, or sequence of events, or choice of events, if appropriate.
- Indicate termination by an outgoing arrow, to a successful termination *Skip* symbol (circle) or a deadlock *Stop* symbol (square).

Illustration

Figure 3.1 illustrates these components for a process P defined as follows.

$$P = e \rightarrow R$$

$$R = (f \rightarrow g \rightarrow S) \,\square\, (h \rightarrow Q) \,\square\, (j \rightarrow Q)$$

$$S = \mathit{term} \rightarrow \mathit{Skip}$$

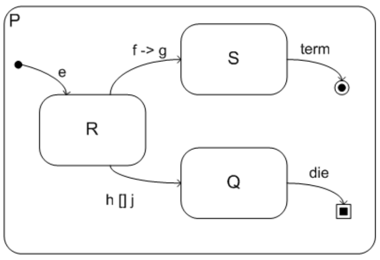

Figure 3.1 CSP Transitions: a sequential process, showing event transitions, states, and *Skip* and *Stop* termination.

$Q = die \rightarrow Stop$

Variant: Interrupts

Communication protocols often require the modelling of interrupts – higher-priority events or transitions that must be serviced before the current event has been fully serviced. Represent interrupts as transitions from outer boxes (the Harel convention).

For example, [Schneider 2000, section 3.4] specifies a writing protocol with an interrupt:

process
$Var = write?x \rightarrow (Var(x) \bigtriangleup Var)$

process
$Var(x) = read!x \rightarrow Var(x)$

The interrupting transition is shown in figure 3.2 as an event arrow from the outer box, having 'higher priority' than the transition from the inner box.

A more complex interrupt structure is modelled in the *Junior*2 specification of [Schneider 2000, section 3.4]:

process
$Junior = Tasks \bigtriangleup x : \{ring, boss\} \rightarrow P(x)$

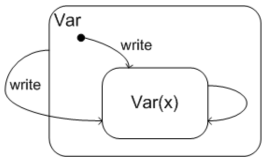

Figure 3.2 CSP Transitions: Schneider's *Var* specification.

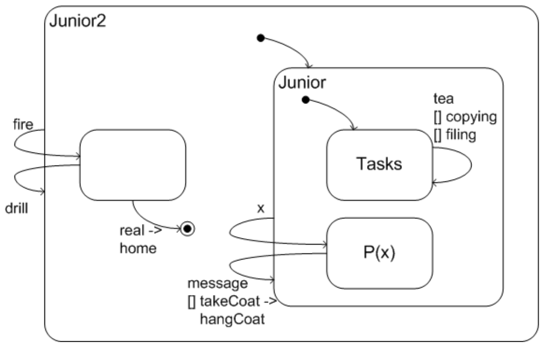

Figure 3.3 CSP Transitions: Schneider's *JUNIOR* specification.

process
$Tasks = tea \rightarrow Tasks \;\square\; copying \rightarrow Tasks \;\square\; filing \rightarrow Tasks$

process
$P(ring) = message \rightarrow Junior$
$P(boss) = takeCoat \rightarrow hangCoat \rightarrow Junior$

process
$$\begin{aligned} Junior2 = Junior \;\triangle\; fire \rightarrow (&real \rightarrow home \rightarrow Skip \\ &\square\; drill \rightarrow Junior2) \end{aligned}$$

The two levels of interrupting transition are clear in figure 3.3.

□

3.2 Processes: communication channels

CSP channels: diagram the structure

Intent

Summarise the structure of a CSP channel communication specification using a diagram.

Problem

CSP specifications contain descriptions of processes communicating with others using a variety of shared synchronised events. It can be difficult to determine the structure of a whole specification by examining the structure of its processes.

Solution

The representation of processes and communication channels is an obvious generalisation of the kind of block diagrams used by CSP authors [Schneider 2000].

- Enclose each sequential process in a rectangular box.
- Draw replicated processes with overlapped boxes.
- Indicate the communication channels as lines between communicating process boxes, with arrows if the communication has a direction (rather than being just a synchronisation event).
- Draw hidden channels inside the box representing the process that hides them. Extend externally visible channels outside the process box.
- By default, do not distinguish replicated channels from single channels: the CSP gives the details.

Illustration

Figure 3.4 illustrates these components for a process *OuterProc* defined as

$$OuterProc = (\|_n \; Proc(n) \; \| \; Proc2) \setminus \{mid\}$$

where the $Proc(i)$ have channels *in* and *mid*, and *Proc*2 has channels *mid* and *out*.

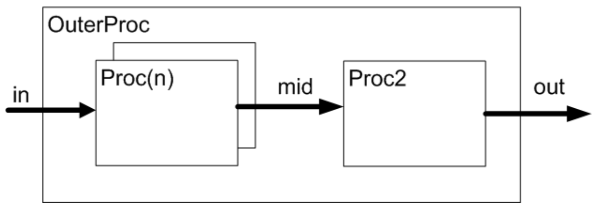

Figure 3.4 CSP Channels: parallel communicating processes, showing replicated processes and channels.

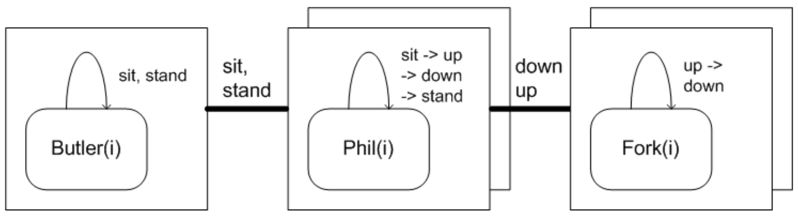

Figure 3.5 CSP Channels: a Dining philosophers specification. The philosopher and fork processes are replicated.

Variants: Replication

- Indicate replicated channels with multiple heads or tails, if necessary (not illustrated).
- 'Explode' replicated or labelled processes to show the constituent parts, if necessary. For example, figure 3.5 shows the well-known dining philosophers example [Hoare 1985, section 2.5], drawn with replicated processes; figure 3.6 shows an exploded variant.

□

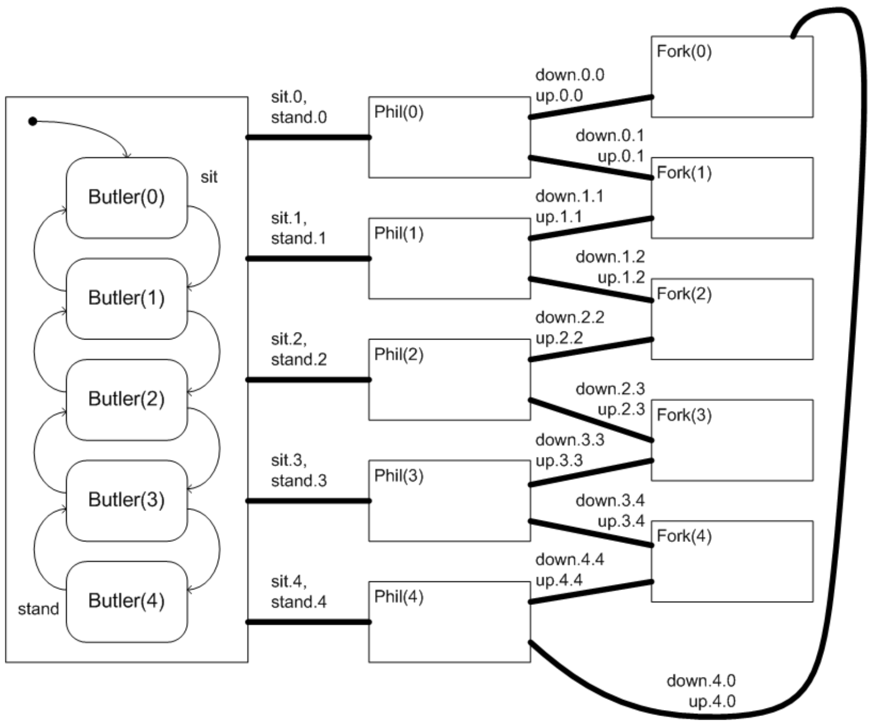

Figure 3.6 CSP Channels: exploded variant of the Dining philosophers specification. The philosopher and fork processes are unfolded into separate processes.

3.3 Processes: full diagrams

CSP: diagram the structure

Intent

Summarise the structure of a CSP specification using a diagram.

Problem

CSP specifications contain descriptions of various processes, each with its own internal behaviour, each communicating with certain others on a variety of shared synchronised events. It can be difficult to determine the structure of a whole specification from the structure of the processes and their parts.

Solution

- Use CSP Transitions: diagram the structure for the internal structure of a single process
- Use CSP Channels: diagram the structure for the communication structure between parallel processes (enclose the relevant process CSP Transitions diagram in the communicating process rectangular box)
- When two specifications have a refinement relationship between them, draw the individual specifications as before, and indicate the refinement relationship by position on the page, by drawing the more abstract specification above the more concrete specification, and drawing the appropriate refinement symbol between them
- If it is necessary to emphasise the state transitions over the communications, reverse the weights of the arrows.

Illustration

Figure 3.7 illustrates the CSP Purse specification from [Srivatanakul *et al.* 2003], given in appendix B.

An exploded variant of the Purse diagram, figure 3.8, shows how the various substates engage in the parallel composition.

□

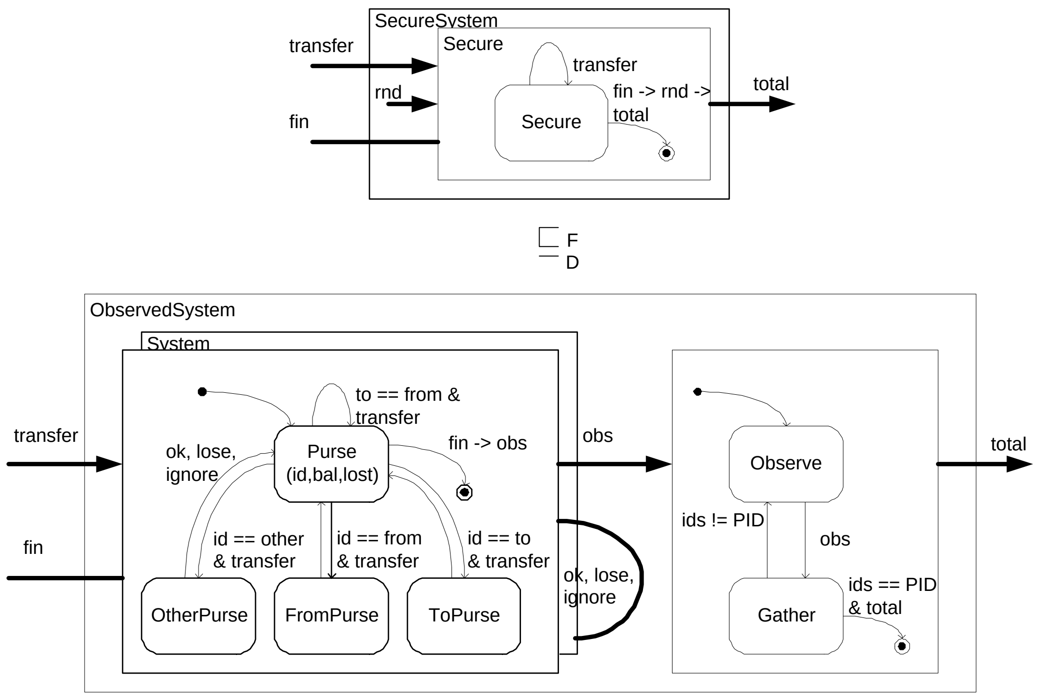

Figure 3.7 CSP diagram: the *Purse* specification. Sequential processes are shown as state transition diagrams; parallel communication channels are bold arrows; refinement is shown by relative position

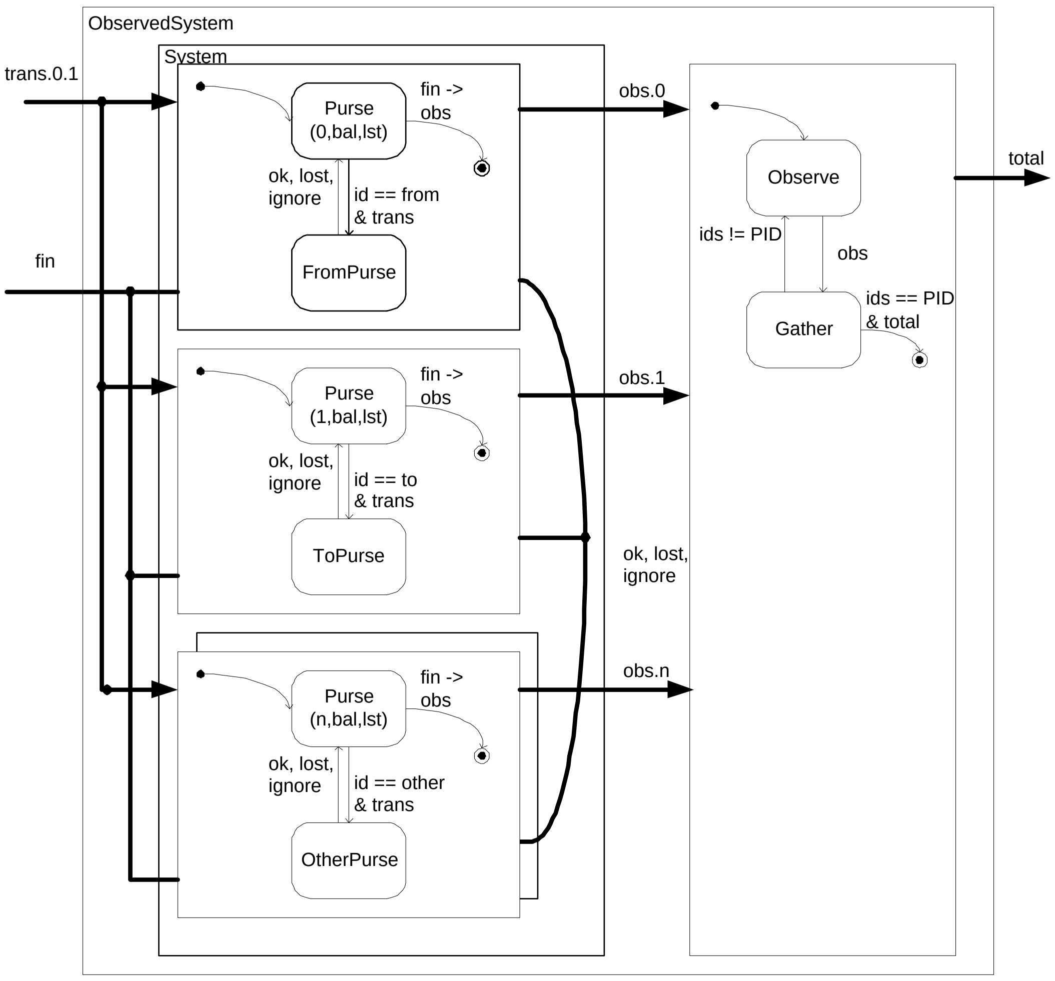

Figure 3.8 CSP diagram: the refined CSP *Purse* specification, exploded view. The Purse state is unfolded into separate parallel processes; the substates are shown in the appropriate process.

Chapter 4

Application of Diagram Patterns to Z

The Z notation is similar in origin to B, but does not have the constraining influence of mechanised proof obligation generation. Z is used to produce precise but flexible specifications, that can support refinement and proof if needed. Implicit patterns of use have emerged in Z texts [Spivey 1992], [Barden *et al.* 1994], and in tool implementations.

4.1 Z graphical notation

The Z formal notation already includes a graphical component, in that certain chunks of specification (vertical schemas) are enclosed in boxes:

```
─ SchemaName ──────────────────────────
│ declaration1
│ declaration2
│ ...
├──────────
│ predicate1
│ predicate2
│ ...
└──────────────────────────────────────
```

This notation aids the readability of Z, by making the separation between the schema items obvious. However, the relationship between the items is not very clear. The diagram patterns are designed to overcome this limitation for a variety of ways of defining items and their relationships in Z.

4.2 The Delta/Xi pattern

The first Z pattern provides diagrammatic summaries of the most common style of Z specification, the state-and-operations or Delta/Xi style. This comprises schemas defining the state (as above). Operations are also defined in schemas which use the names of state schemas, prefixed with Δ (if the operation can change the state) or Ξ (if the operation does not change the state). The prefix indicates the introduction of before- and after-states, and, in the case of Ξ, predicates equating the components of the before state to those of the after state. In addition, any named schema can be included in other schemas (declarations are concatenated; predicates are conjoined), and can be used in most Z structures requiring a mathematical set or relation.

The Delta/Xi pattern underlies many other patterns, including promotion and refinement patterns [Stepney *et al.* 2003b], [Stepney *et al.* 2003c]. Some of these give rise to variants, or to separate diagram patterns.

Delta/Xi : diagram the structure

Intent

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

Problem

A Z specification is written 'bottom-up' (declaration before use) and can be factored into many pieces. Components are included in various other components, so that it becomes difficult to see the overall structure.

Solution

The diagram records the structure of the state and operation schemas, highlighting any Delta/Xi-related patterns used. The notation is adapted and extended from that used by [d'Inverno & Luck 2001].

Use different polygons to distinguish kinds of schema purpose:

- Draw state schemas as named rectangles
- Draw operation schemas as named hexagons
- Draw other data types as named parallelograms
- Where appropriate for clarity, use extra structuring schemas not defined in the specification. 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.)

Use arrows to represent structural links:

- 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 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
  - for clarity, elide an arrow directly to a box if there is an alternative path to that box
- Indicate other relationships among schemas by dashed lines from the referring construct 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 related or derived 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

Use a positional convention. Where possible, without distorting the diagram, draw inclusion arrows upwards, so that the simplest schemas are at the top of the diagram, and constructs that are more complex are further down the page. (This is to help the reader; position is not intended to express any structural information.)

Illustration

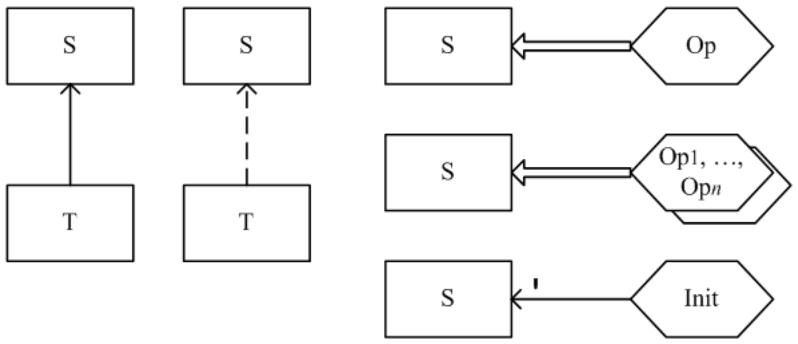

Figure 4.1 Z Delta/Xi diagrams: (a) schema T includes schema S, (b) schema T references schema S (c.1) operation *Op* includes ΔS, (c.2) several operations Op_i include ΔS, (c.3) initialisation operation *Init* includes $S\,'$

Figure 4.1 shows the pattern representations of the commonest structural components of Delta/Xi Z specifications:

- First, schema T includes schema S, either as declaration as $T == [\,S;\ \ldots \mid \ldots]$, or as a predicate as $T == [\ldots \mid S \land \ldots]$.
- Then schema T refers to schema S other than by inclusion, for example as $T == [\,f : x \pfun S \ldots \mid \ldots]$.

The three operation illustrations are:

- operation schema *Op* includes schema S as $Op == [\,\Delta S \ldots \mid \ldots]$;
- multiple schemas Op_i have precisely the same relationships with other schemas, so their names can be listed in the same box, thereby drawing attention to their similar structures;
- an initialisation schema *Init* includes schema S as $Init == [\,S\,' \ldots \mid \ldots]$.

Appendix C, figures C.1–C.4, give an example of the use of the diagrams to express the structure of a large specification, the purse [Stepney *et al.* 2003c].

Variants

- The notation can be extended to show conjectures.

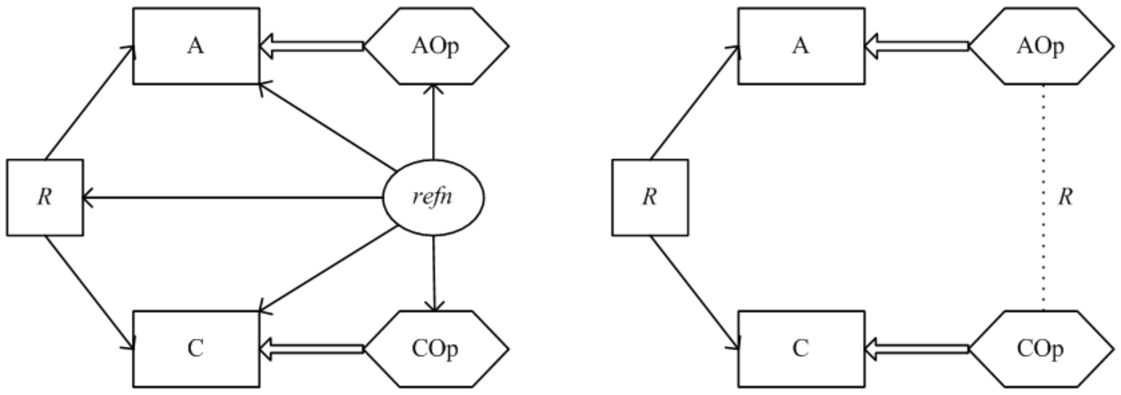

Figure 4.2 Z Delta/Xi diagrams: (a) The refinement pattern in full, showing the refinement relation R and the refinement conjecture; (b) the abbreviated pattern, showing the refinement relation, and the conjecture replaced by a dotted line between abstract and concrete operations

  - Draw a conjecture in an oval, labelled with a suitable name, pointing to any referenced schemas (see Appendix C, figure C.2), or the refinement conjecture in Figure 4.2.
- The notation can also be extended to show only the refinement relation, for the refinement pattern elaboration.
  - Elide the refinement conjecture, and use a dotted line to indicate the relationship between abstract and concrete operations (figure 4.2).
- For large specifications, a diagram may be split into sub-diagrams for clarity, perhaps using a separate diagram for each operation, or family of operations, reproducing relevant parts of the diagram.
  - Represent schemas or other data type boxes occurring in more than one sub-diagram as rounded corners drawn inside the boxes (see Appendix C).

□

4.3 The Delta/Xi pattern, State Transition viewpoint

Occasionally it is useful to diagram the implicit underlying state transition relation in a Delta/Xi pattern specification. A similar diagram to the CSP Transitions diagram is used.

Delta/Xi Transitions: diagram the structure

Intent

Summarise the state transition structure of a Delta/Xi specification using a diagram.

Problem

The Delta/Xi pattern defines a state and operations on that state. This can be considered as a *state transition relation*, although the actual sub-states and transitions are usually implicit. Sometimes they are not, and it can be useful to draw a diagram of the actual state transition relations.

Solution

The (sub-)states, and the state transitions, governed by operations, are represented as follows.

- Represent states and sub-states in rounded boxes, labelled with the schema name of the state, or the predicate distinguishing the sub-state. Nest sub-states inside their states.
- Represent transitions with arrows between states. Draw a transition that may occur from several sub-states (that is, an operation whose precondition is *true* in several sub-states) from the largest including sub-state. Label the arrow with an operation, sequence of operations, or choice of operations, as appropriate.
- Indicate initialisation to the initial state by an incoming arrow, if necessary.
- Indicate finalisation by an outgoing arrow to a termination symbol (circle).

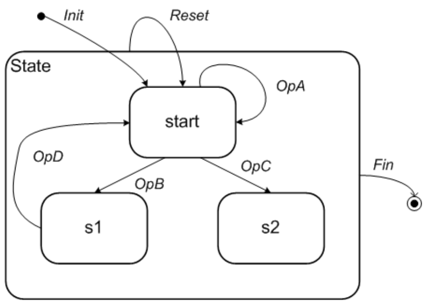

Figure 4.3 Z Delta/Xi Transitions diagram: showing operation transitions and sub-states, initialisation, and finalisation

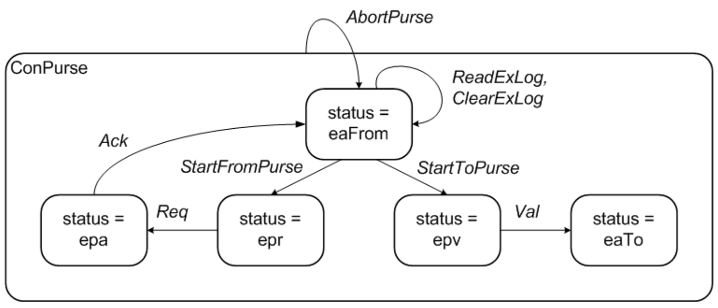

Figure 4.4 Z Delta/Xi Transitions diagram: the concrete unpromoted *Purse*.

Illustration

Figure 4.3 shows the basic notations.

[Stepney *et al.* 2000] gives a Z specification of an electronic Purse and its refinement. The concrete unpromoted Purse is diagrammed in figure 4.4.

Related Pattern

CSP Transition: diagram the pattern, section 3.1, above.

□

4.4 The Promotion pattern

Simple Z state and operation specifications can be combined into a system specification using *promotion*. Promotion is a special case of the Delta/Xi pattern, and is described in [Stepney *et al.* 2003b].

Promotion : diagram the structure

Intent

Summarise the structure of the promotion pattern using a diagram.

Problem

Using Z promotion pattern adds a certain amount of surface complexity to the specification, which can make it hard to follow.

Solution

Use the layout illustrated in figure 4.5 to separate the local state and operations, the framing (Φ) schemas, and the global system state and operations.

Illustration

Figure 4.5 shows two instances of promotion. The first shows the structure for update operations; the second shows the special form for initialisation (creation of new local instances).

Promotion is used in the Purse diagrams, Appendix C – figure C.1 summarises the entire *Purse* specification structure, exposing the promotion structure; figure C.4 details the concrete specification, highlighting the promotion components.

Related Pattern

Delta/Xi: diagram the pattern provides the basic notations.

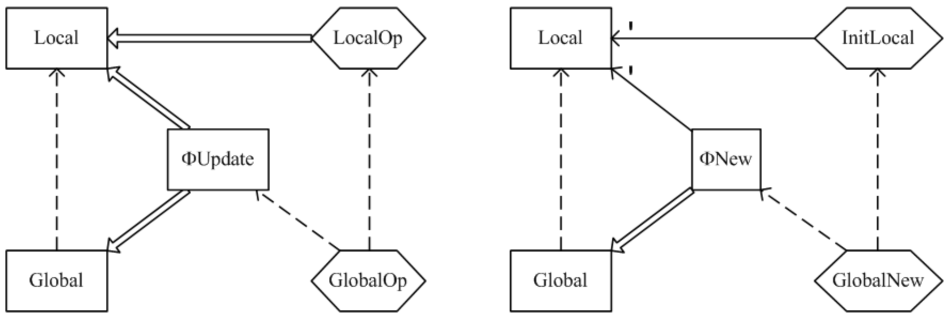

Figure 4.5 Z diagram: (a) structure of the promotion pattern (b) structure of the promotion pattern, for the initialisation operation

---

□

4.5 The Morph pattern

The Morph pattern could be used for any functional specification – translations, mappings, functions. Some diagrams already exist for morphs, for instance, Cleanroom boxes [Prowell *et al.* 1999] represent functions and function decomposition. For Z morphs, we use a variant of these notations, focusing on the component functions and the data inputs and outputs. The decomposition is static, unlike the evolving Cleanroom diagrams. Morph diagrams provide visual assurance that the system-level stimulus/response mapping is preserved by the combination of the lower level functions.

Z Morph : diagram the structure

Intent

Summarise the structure of a Z specification that uses the Morph pattern using a diagram.

Problem

A morph, such as a compiler specification, is a mapping that itself comprises many lower-level mappings. The bottom-up style of Z gives clear expression to low-level mappings, but the relationship of each low-level component to the whole is concealed in the detail.

Solution

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

Represent functions by polygons:

- draw the highest level mapping or function of interest as a grey background rectangle
- draw named component 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).

Represent inputs (stimuli) and outputs (responses) by arrows:

- draw input arrows into the box; if the input is a constant, draw it emanating from a constant source (indicated by a blob)
- draw output 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. Decorate line junctions representing duplication or separation of component input items:

- use an 'exploding star' symbol to 'explode' a product type input into its components
- use a 'plus' symbol to copy (fork) an argument into several different functions

Use a positional convention as much as possible so that, without distorting the diagram, the arguments flow from left to right. First, minimise the number of crossing lines, and then as far as possible have arguments entering boxes in the correct order.

Illustration

Figures D.1 and D.2 (Appendix D) show Z morph diagrams for two functions in a compiler specification.

Variants

The notation can capture special features of functional specification. For example, curried functions can be drawn as nested functions.

□

Chapter 5

Application of Diagram Patterns to Circus

Circus [Cavalcanti *et al.* 2002], [Woodcock & Cavalcanti 2002] is a combination of Z and CSP, with a strict formal semantics in the style of Unified Programming [Hoare & He 1998]. One Circus style describes the state transitions in Z (Delta/Xi), and communication between processes as CSP events. When Circus is used in this style, the diagram pattern comprises the appropriate CSP diagram patterns and Delta/Xi: diagram the structure.

Circus: diagram the structure

Intent

Summarise the structure of a Circus specification using a diagram.

Problem

Circus specifications contain descriptions of various processes, each with its own internal state and behaviour, each communicating with certain others on a variety of shared synchronised events. It can be difficult to determine the structure of a whole specification from examining the structure of its parts.

Solution

There are two kinds of structure to a Circus specification.

- For a single sequential process, the state transition structure is specified as a Z Delta/Xi style. Use the corresponding Delta/Xi : diagram the structure pattern.

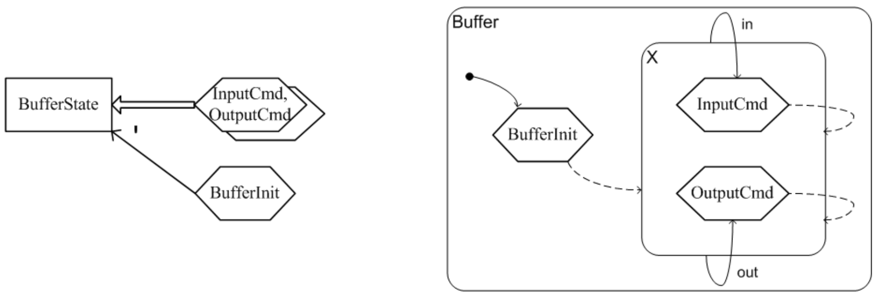

Figure 5.1 Circus diagram: the *Buffer* specification. The state and operations structure is shown using Z Delta/Xi diagrams; the transitions are shown using CSP diagrams, with the relevant Z operation.

- For a collection of sequential processes communicating by shared events on named channels, the communications are specified in CSP style. Use the corresponding part of the CSP : diagram the structure pattern.

Illustration

[Cavalcanti *et al.* 2002] use Circus to specify a buffer. A diagram of this specification is shown in figure 5.1.

Constraint

This diagram pattern is appropriate only when the Circus specification is written using the relevant Z pattern. (Other patterns for Circus specifications are being developed.)

□

Chapter 6

Application of Diagram Patterns to CSP| |B

CSP| |B (Communicating B machine) [Treharne & Schneider 2000], [Schneider & Treharne 2002], [Treharne *et al.* 2003] is an informal combination of B and CSP. It describes the state transitions in B, and communication between processes in CSP.

The CSP| |B diagram pattern combines the appropriate CSP diagram pattern and the B diagram pattern.

CSP| |B: diagram the structure

Intent

Summarise the structure of a CSP| |B specification using a diagram.

Problem

CSP| |B specifications contain descriptions of several B machines, each with its own internal state and behaviour, communicating with certain others on a variety of shared synchronised events. It can be difficult to determine the structure of a whole specification from examining the structure of its parts.

Solution

There are two kinds of structure to a CSP| |B specification.

- For a single sequential process, the state transition structure is specified as a B machine. Use the B: diagram the structure pattern.

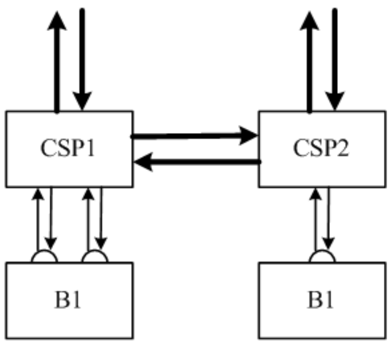

Figure 6.1 CSP| |B diagram: a combination of the CSP and B diagrams.

- For a collection of sequential processes communicating by shared events on named channels, the communications are specified in CSP style. Use the corresponding CSP : diagram the structure pattern.

Illustrations

The basic notation is illustrated in figure 6.1.

The "Bank Counter" CSP| |B specification from [Treharne *et al.* 2003] is diagrammed in figure 6.2.

□

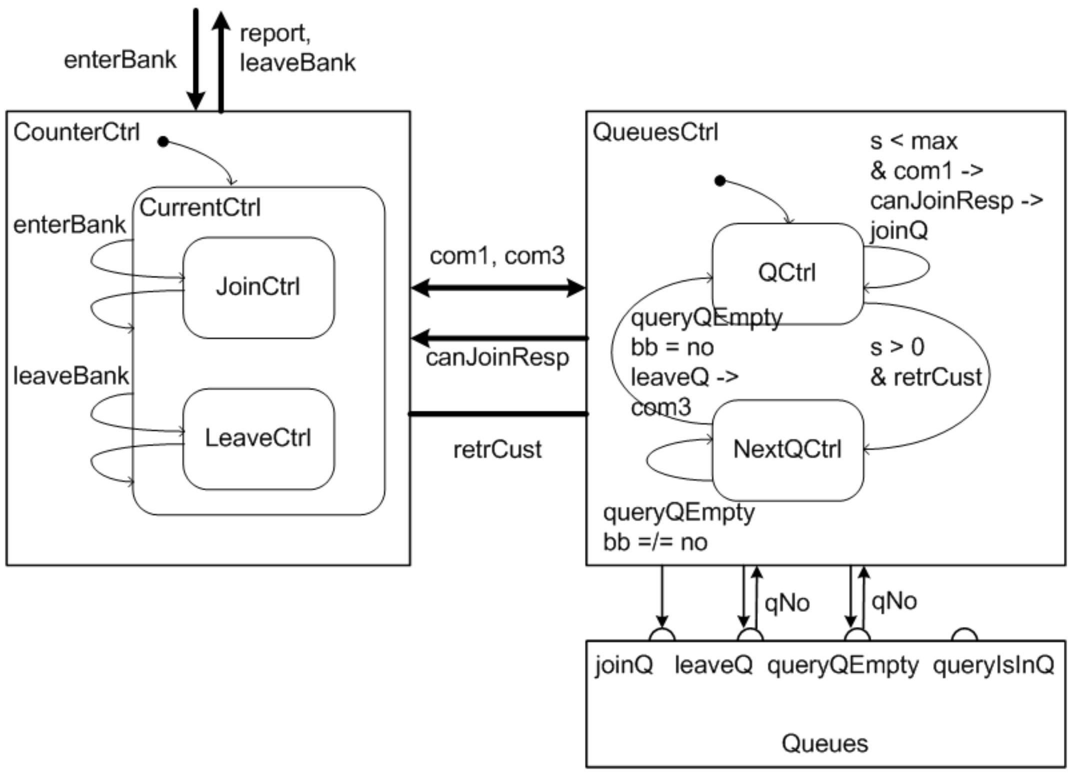

Figure 6.2 CSP||B diagram: the Bank Counter specification. Sequential processes are shown as B machine diagrams; parallel communication channels are shown as CSP bold arrows.

Chapter 7

Conclusions

This paper presents diagrammatic patterns for a number of well-known and recent formal notations and conventions. We show how the basic structure of a formal specification can be captured and emphasised. The ability of some of these patterns to illustrate the structure of one of the largest Z specifications published to date (appendix C) gives us confidence in their scalability.

We have also shown how more advanced concepts can be added, either by extending existing diagram notations, or by combining patterns. The ability to combine the separate diagram patterns for combined languages (as for Circus and CSP||B) gives us confidence that these patterns are robust.

The diagram patterns are defined according to the Design the Diagram meta-pattern. This gives a consistent style of presentation, and assists in the construction of catalogues and other pattern support. Our diagram patterns are derived from diagrams in the literature, and the kinds of diagrams people sketch when explaining, or trying to understand, specifications.

An important feature of Diagram the structure patterns is that these are informal diagrams, not an alternative formalism. Consistency of notational usage is not generally of great import, so long as the required structural highlighting is achieved. Completeness is usually not needed: completing a diagrammatic representation with "clutter" hides the structure rather than emphasising it.

We have drawn our diagrams from existing formal specifications. However, the patterns are intended for use during specification development. Diagrams drawn *post hoc* may not look as elegant as ones prepared as part of the development, but they may be used to indicate refactoring opportunities.

Diagrams can be used in many ways, but are essentially intended to be part of the documentation of a formal development. One could envisage formal support tools producing summary diagrams (as is already done in, for example, Microsoft

Access's "relationships view" of the conceptual structure of a database); however, the contribution of diagram patterns to formal tool support is far out-weighed by the importance of the formal patterns themselves.

Acknowledgements

We gratefully acknowledge the involvement of Steve Schneider and Jim Woodcock in commenting on our diagramming of their B, CSP, Circus and CSP||B specifications. Regine Laleau provided useful comment on the B notations, and on other potential applications of these diagrams. Steve King and Richard Paige have provided checking and encouragement. We are also most grateful for the work of our various colleagues in the research underlying the Purse specifications and the Z patterns.

Appendix A

Bibliography

[Abrial 1996]
J-R. Abrial. *The B-Book.* CUP, 1996.

[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.

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

[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, France, January 2002*, volume 2272 of *LNCS*. Springer, 2002.

[Bert *et al.* 2003]
Didier Bert, Jonathan P. Bowen, Steve King, and Marina Walden, editors. *ZB2003: Third International Conference of B and Z Users, Turku, Finland*, volume 2651 of *LNCS*. Springer, 2003.

[Booch *et al.* 1999]
Grady Booch, James Rumbaugh, and Ivar Jacobson. *The Unified Modeling Language User Guide.* Addison-Wesley, 1999.

[Brooke & Paige 2002]
P. J. Brooke and R. F. Paige. The design of a tool-supported graphical notation for timed csp. In *Proc. Integrated Formal Methods 2002 (IFM'02)*, volume 2335 of *LNCS*. Springer, 2002.

[Cavalcanti *et al.* 2002]
A. L. C. Cavalcanti, A. C. A. Sampaio, and J. C. P. Woodcock. Refinement of actions in Circus. In *Proceedings of REFINE'2002*, Electronic Notes in Theoretical Computer Science, 2002. Invited Paper.

[CCTA 1990]
CCTA. *SSADM Version 4 Reference Manual.* NCC Blackwell Ltd, 1990.

[Coplien & Schmidt 1995]
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, 2001.

[Facon *et al.* 2000]
P. Facon, Regine Laleau, and H. P. Nguyen. From OMT diagrams to B specifications. In M. Frappier and H. Habrias, editors, *Software Specification Methods. An Overview Using a Case Study*, pages 57–77. Springer, 2000.

[FDR 2000]
Failure-Divergence Refinement - FDR2 User Manual. Formal Systems (Europe) Ltd., 2000.

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

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

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

[Goodland & Slater 1995]
M. Goodland and C. Slater. *SSADM Version 4: A Practical Approach.* McGraw-Hill, 1995.

[Harel 1987]
D. Harel. Statecharts: A visual formalism for complex systems. *Science of*

Computer Programming, 8(3):231–274, June 1987.

[Hoare & He 1998]
C. A. R. Hoare and Jifeng He. *Unified Theories of Programming.* Prentice Hall, 1998.

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

[Laleau 2002]
Regine Laleau. Conception et développement formels d'applications bases de données. Thèse CEDRIC, habilitation à diriger des recherches, Université d'Evry, 2002.

[Martin *et al.* 1998]
Robert C. Martin, Dirk Riehle, and Frank Buschmann, editors. *Pattern Languages of Program Design 3.* Addison-Wesley, 1998.

[Meszaros & Doble 1998]
Gerard Meszaros and Jim Doble. A pattern language for pattern writing. In [Martin *et al.* 1998].

[Polack 2003]
Fiona Polack. Exploring the informal translation of OMT object models in B. Technical Report YCS-2003-351, University of York, 2003.

[Prowell *et al.* 1999]
S. J. Prowell, C. J. Trammell, R. C. Linger, and L. H. Poore. *Cleanroom software engineering: technology and process.* Addison-Wesley, 1999.

[Roscoe 1994]
A. W. Roscoe. Model-checking CSP. In *A classical mind: essays in honour of C. A. R. Hoare*, pages 353–378. Prentice Hall, 1994.

[Roscoe 1997]
A. W. Roscoe. *The Theory and Practice of Concurrency.* Prentice Hall, 1997.

[Schneider & Treharne 2002]
Steve Schneider and Helen Treharne. Communicating B Machines. In [Bert *et al.* 2002].

[Schneider 2000]

Steve Schneider. *Concurrent and Real-time Systems: the CSP approach.* Wiley, 2000.

[Schneider 2001]
Steve Schneider. *The B-Method: an introduction.* Palgrave, 2001.

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

[Srivatanakul *et al.* 2003]
Thitima Srivatanakul, John A. Clark, Susan Stepney, and Fiona Polack. Challenging formal specifications by mutation: a CSP security example. In *APSEC-2003: 10th Asia-Pacific Software Engineering Conference, Chiang Mai, Thailand, December, 2003.* IEEE, 2003.

[Stepney & Nabney 2003]
Susan Stepney and Ian T. Nabney. The DeCCo papers. Technical Reports YCS-2003-358–363, University of York, 2003.

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

[Stepney *et al.* 2002]
Susan Stepney, Fiona Polack, and Ian Toyn. Refactoring in maintenance and development of Z specifications and proofs. In *REFINE 2002, BCS-FACS Refinement Workshop, Copenhagen, Denmark, July 2002*, volume 70(3) of *ENTCS*. Elsevier, 2002.

[Stepney *et al.* 2003a]
Susan Stepney, Fiona Polack, and Ian Toyn. An outline pattern language for Z: five illustrations and two tables. In [Bert *et al.* 2003], pages 2–19.

[Stepney *et al.* 2003b]
Susan Stepney, Fiona Polack, and Ian Toyn. Patterns to guide practical refactoring: examples targetting promotion in Z. In [Bert *et al.* 2003], pages 20–39.

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

versity of York, January 2003.

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

[Treharne & Schneider 2000]
Helen Treharne and Steve Schneider. How to drive a B Machine. In Jonathan P. Bowen, Steve Dunne, Andy Galloway, and Steve King, editors, *ZB2000: First International Conference of B and Z Users, York, UK, August 2000*, volume 1878 of *LNCS*. Springer, 2000.

[Treharne *et al.* 2003]
Helen Treharne, Steve Schneider, and Marchia Bramble. Composing specifications using communication. In [Bert *et al.* 2003].

[Valentine *et al.* 2004]
Samuel H. Valentine, Susan Stepney, and Ian Toyn. A Z patterns catalogue II: definitions and laws, v0.1. Technical Report YCS-2004-383, Department of Computer Science, University of York, October 2004.

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

[Yourdon 1989]
E. Yourdon. *Modern Structured Analysis.* Prentice-Hall, 1989.

Appendix B

CSP Example

Here we provide the CSP specification underlying figure 3.7 (reproduced from [Srivatanakul *et al.* 2003]).

First we introduce some useful constants.

$\mathit{Npurse} = 4$	[number of purses (variable)]
$\mathit{InitBal} = 1$	[initial balance in each purse (variable)]
$\mathit{TotalBal} = \mathit{Npurse} * \mathit{InitBal}$	[hence, total system balance]
$\mathit{PID} = 0\,..\,(\mathit{Npurse} - 1)$	[purse identifiers]
$\mathit{VALUE} = 0\,..\,\mathit{TotalBal}$	[value transferred between purses]

Channel *fin* signals the system to finalise. Channel *transfer* identifies the transfer action, labelled by from purse, to purse, and transfer value. Channel *total* outputs the total value stored and lost in the system, on finalisation. The internal channel *rnd* is used to generate a random value that is considered lost.

```
channel
  $\mathit{fin};\ \mathit{transfer} : \mathit{PID}.\mathit{PID}.\mathit{VALUE};\ \mathit{total} : \mathit{VALUE}.\mathit{VALUE}$
  $\mathit{rnd} : \mathit{VALUE}$
```

A system is *Secure* if, on finalisation, the total value stored is no greater than the initial balance ('no value created'), and if the values stored and lost together equal the initial balance ('all value accounted').

```
process
  $\mathit{Secure} =$
      $\mathit{transfer}?\mathit{from}.\mathit{to}.\mathit{val} \rightarrow \mathit{Secure}$
      $\Box\ \mathit{fin} \rightarrow \mathit{rnd}?v \rightarrow \mathit{total}!(\mathit{TotalBal} - v).v \rightarrow \mathit{Skip}$
```

process

$SecureSystem = Secure \setminus \{| rnd |\}$

In the purse system specification, the channel *obs* is used to observe the *bal*ance and *lost* components of each purse on finalisation. The channels *ok*, *lose*, and *ignore* synchronise the purses on the particular choice of transfer operation.

channel

$obs : PID.VALUE.VALUE;\ ok;\ lose;\ ignore$

When a *Purse* engages in a *fin* event, it outputs its state, then terminates. When it engages in a *transfer* event, it behaves like a *FromPurse*, a *ToPurse*, or an *OtherPurse*, depending on its *id*.

```
$Purse(id, bal, lost) =$
    $fin \rightarrow obs!id.bal.lost \rightarrow Skip$
    $\Box\ transfer?from.to.val \rightarrow$
        if $to = from$ then $Purse(id, bal, lost)$
        else if $(id = from)$ then $FromPurse(id, bal, lost, val)$
        else if $(id = to)$ then $ToPurse(id, bal, lost, val)$
        else $OtherPurse(id, bal, lost)$
```

A *FromPurse*, if it has sufficient balance for the requested transfer, non-deterministically chooses to do an *ok* transfer (decrementing its balance), or to *lose* the transfer (decrementing its balance and incrementing its *lost*), or to *ignore* the transfer (leaving its state unchanged). A *ToPurse* increments its balance for an *ok* transfer, otherwise does nothing. An *OtherPurse* does nothing.

```
process
  $FromPurse(id, bal, lost, val) =$
      if $val \leq bal$ then (
          $ok \rightarrow Purse(id, bal - val, lost)$
          $\sqcap\ lose \rightarrow Purse(id, bal - val, lost + val)$
          $\sqcap\ ignore \rightarrow Purse(id, bal, lost)$
      ) else $(ignore \rightarrow Purse(id, bal, lost))$
```

process
$\quad ToPurse(id, bal, lost, val) =$
$\qquad ok \rightarrow Purse(id, bal + val, lost)$
$\qquad \Box\ lose \rightarrow Purse(id, bal, lost)$
$\qquad \Box\ ignore \rightarrow Purse(id, bal, lost)$

process
$\quad OtherPurse(id, bal, lost) =$
$\qquad ok \rightarrow Purse(id, bal, lost)$
$\qquad \Box\ lose \rightarrow Purse(id, bal, lost)$
$\qquad \Box\ ignore \rightarrow Purse(id, bal, lost)$

A purse *System* runs the purses in synchronised parallel, with each purse initially having a zero *lost* component.

process
$\quad System = \|[\{\!| ok, lose, ignore, fin, transfer |\!\}]\|\ id : PID \bullet Purse(id, InitBal, 0)$

The processes *Observe* and *Gather* collect and output the total value of all the purses' balances and lost components on finalisation.

process
$\quad Observe(tb, tl, ids) =$
$\qquad obs?id.bal.lost \rightarrow Gather(tb + bal, tl + lost, ids \cup \{id\})$

process
$\quad Gather(tb, tl, ids) =$
$\qquad \text{if } ids = PID \text{ then } total!tb.tl \rightarrow Skip \text{ else } Observe(tb, tl, ids)$

The *ObservedSystem* is the purse *System* and the initial *Observe* process, with the internal channels hidden.

process
$\quad ObservedSystem =$
$\qquad (System\ |[\ \{\!| obs |\!\}\ ]|\ Observe(0, 0, \varnothing)) \setminus \{\!| obs, ok, lost, ignore |\!\}$

The *ObservedSystem* is a *SecureSystem*: it does not create value, and accounts for all lost value.

assert
$\quad SecureSystem \sqsubseteq_{FD} ObservedSystem$

Appendix C

Z Delta/Xi pattern diagram illustrations

This appendix illustrates the use of the Z Delta/Xi : diagram the structure pattern on a large real-world specification, that of the electronic Purse in [Stepney *et al.* 2000], shown in figures C.1–C.4.

These diagrams makes use of several features:

- using 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)

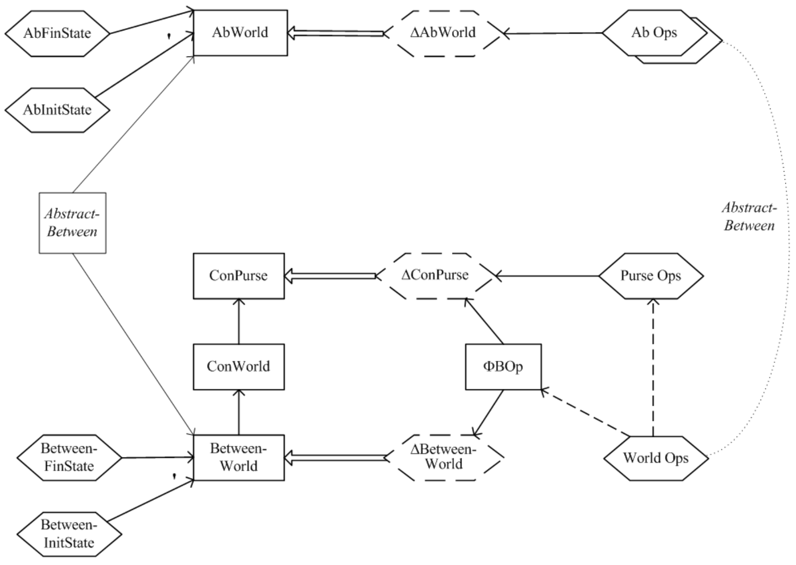

Figure C.1 Z diagram: summary of the structure of the full *Purse* model [Stepney *et al.* 2000].

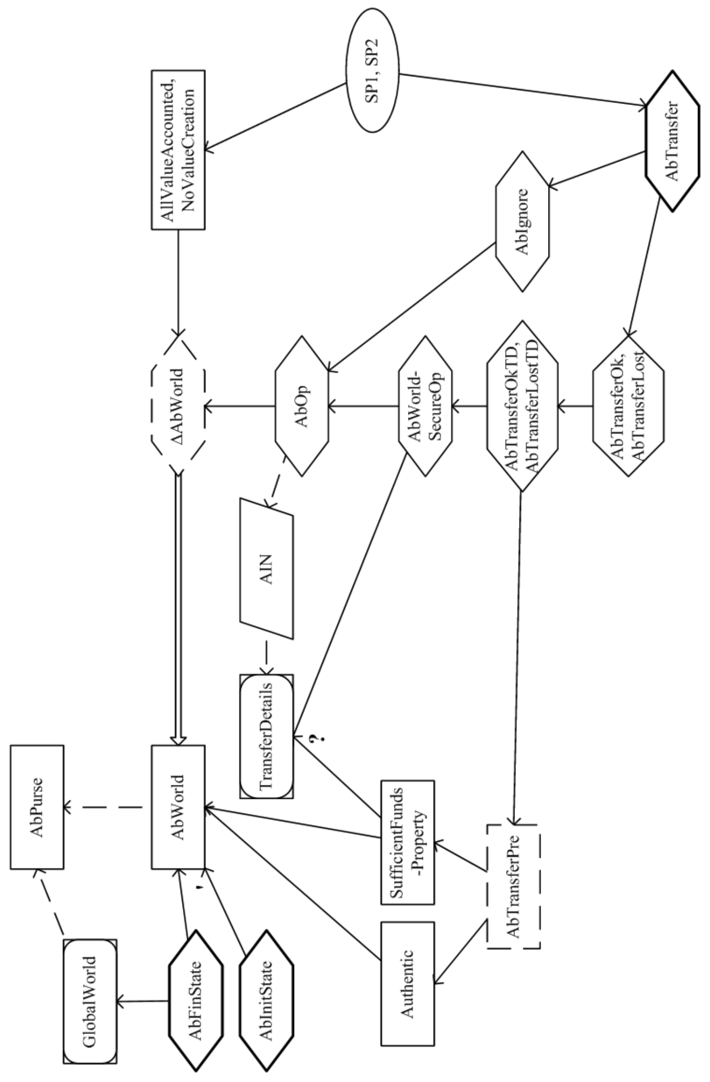

Figure C.2 Z diagram: detail of the structure of the abstract Z *Purse* model [Stepney *et al.* 2000, chapter 3].

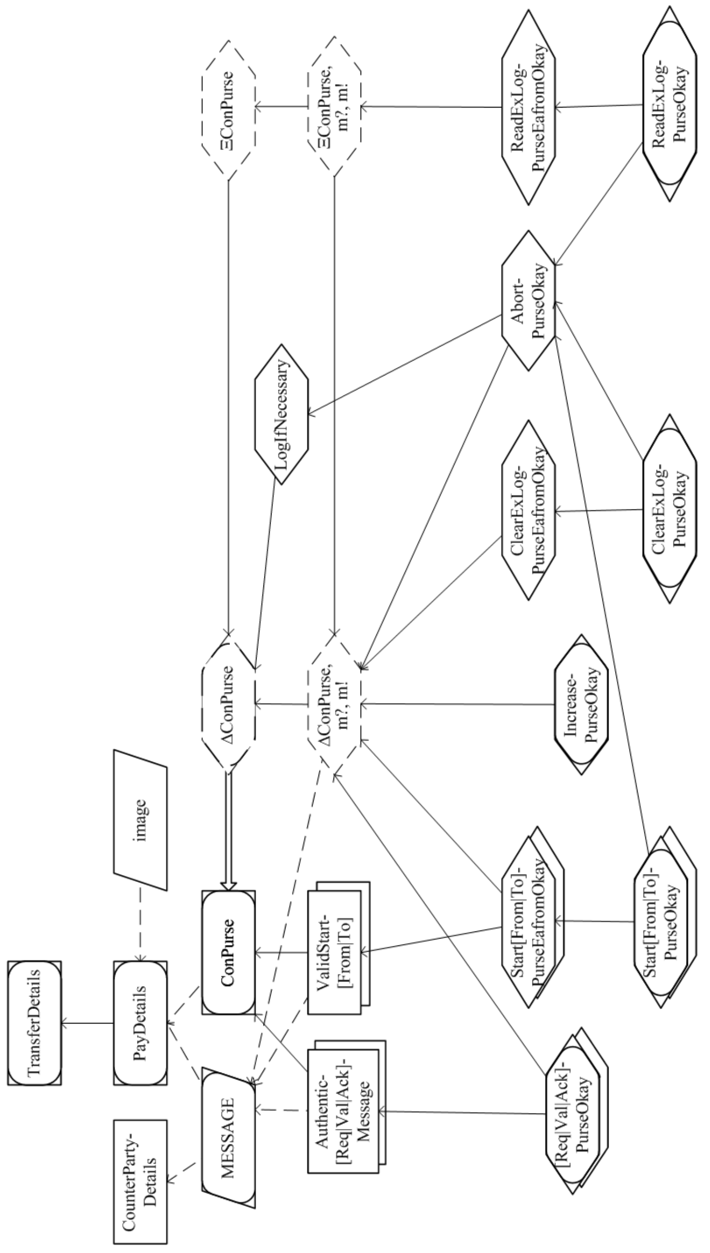

Figure C.3 Z diagram: detail of the structure of the concrete Z *Purse* model, local state [Stepney *et al.* 2000, chapter 4].

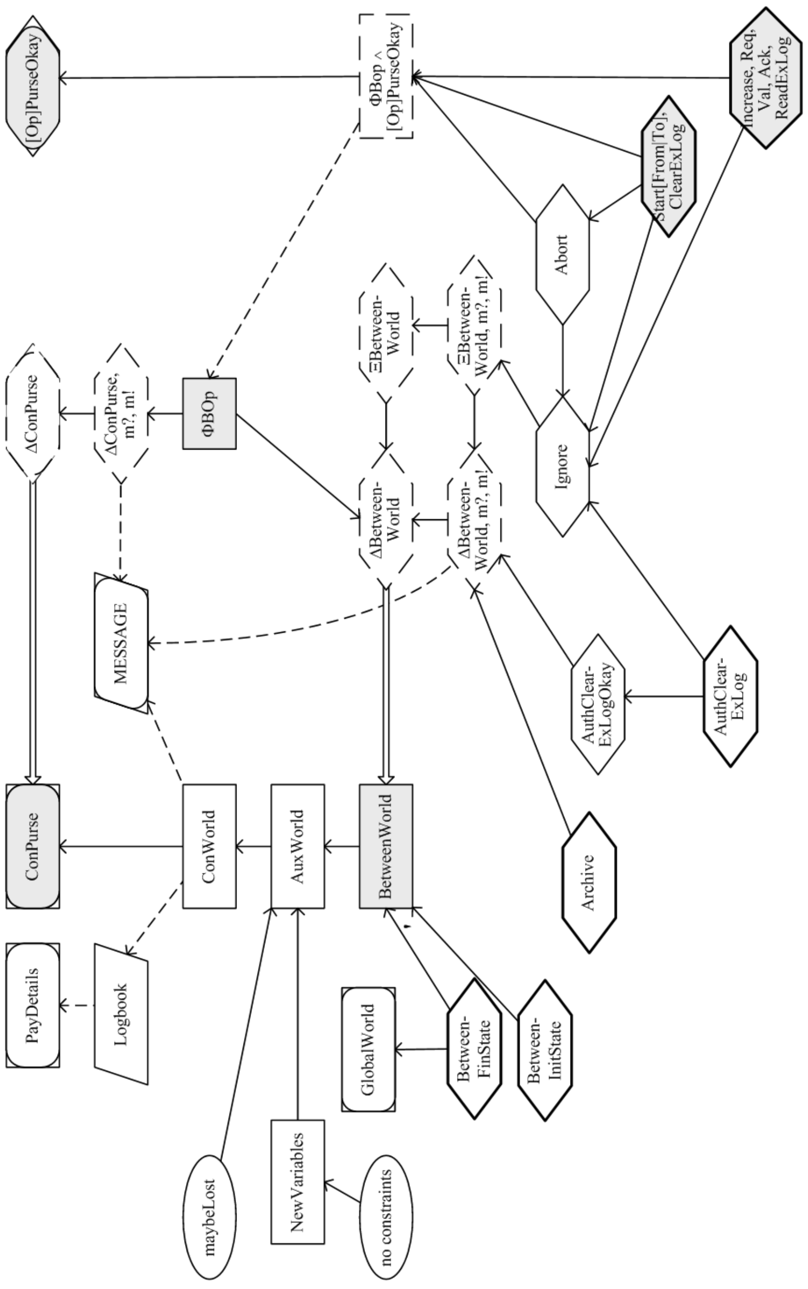

Figure C.4 Z diagram: detail of the structure of the concrete Z *Purse* model, global state [Stepney *et al.* 2000, chapter 5], promotion pattern highlighted.

Appendix D

Z Morph pattern diagram illustrations

This appendix illustrates the use of the Z Morph : diagram the structure pattern on a medium-sized example specification, that of the toy compiler in [Stepney 1993] (figure D.1), and on a large real-world specification, that of the real compiler in [Stepney & Nabney 2003] (figure D.2).

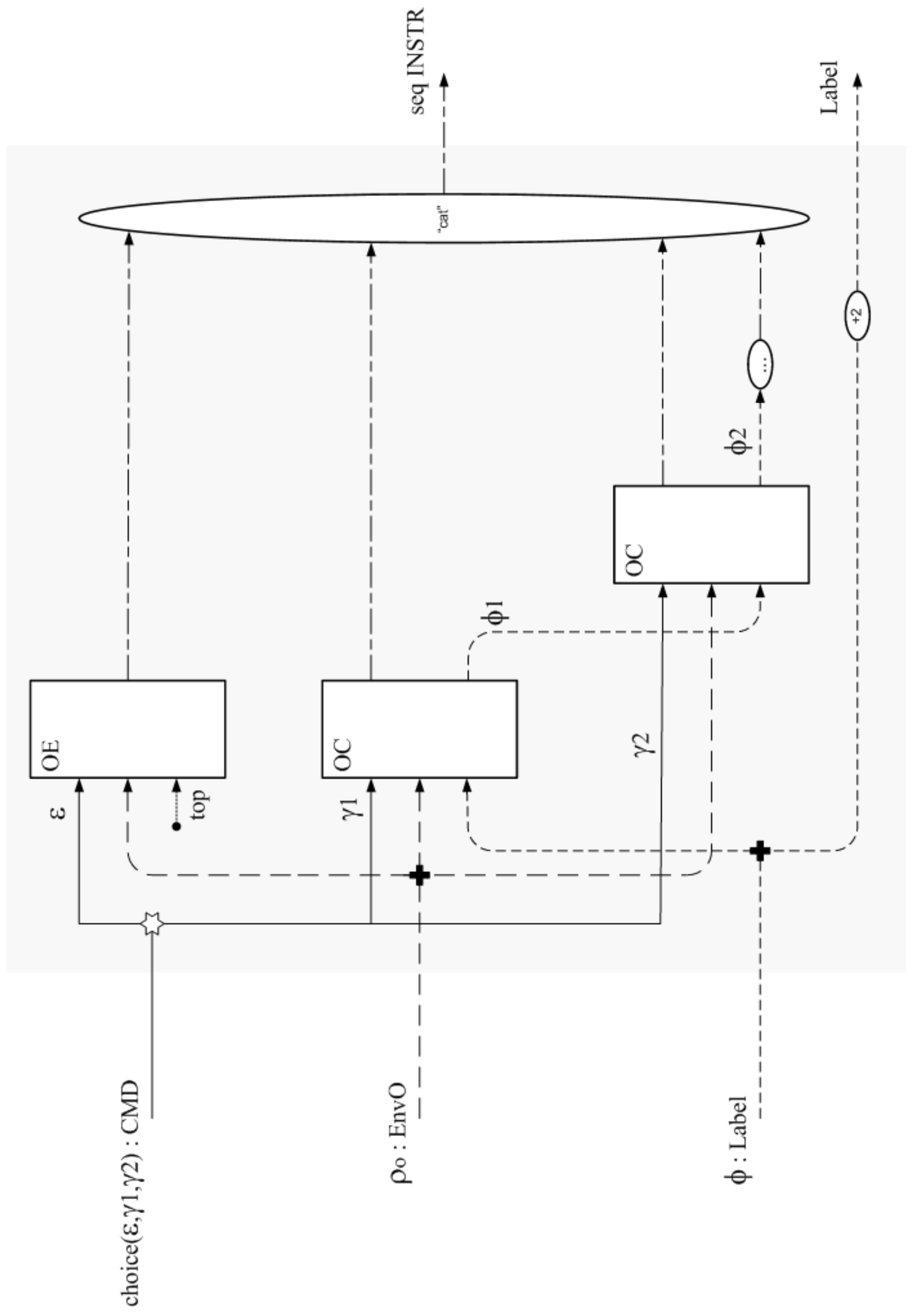

Figure D.1 Z diagram: the structure of the Tosca command compile function, applied to the *choice* command [Stepney 1993, section 11.4.5].

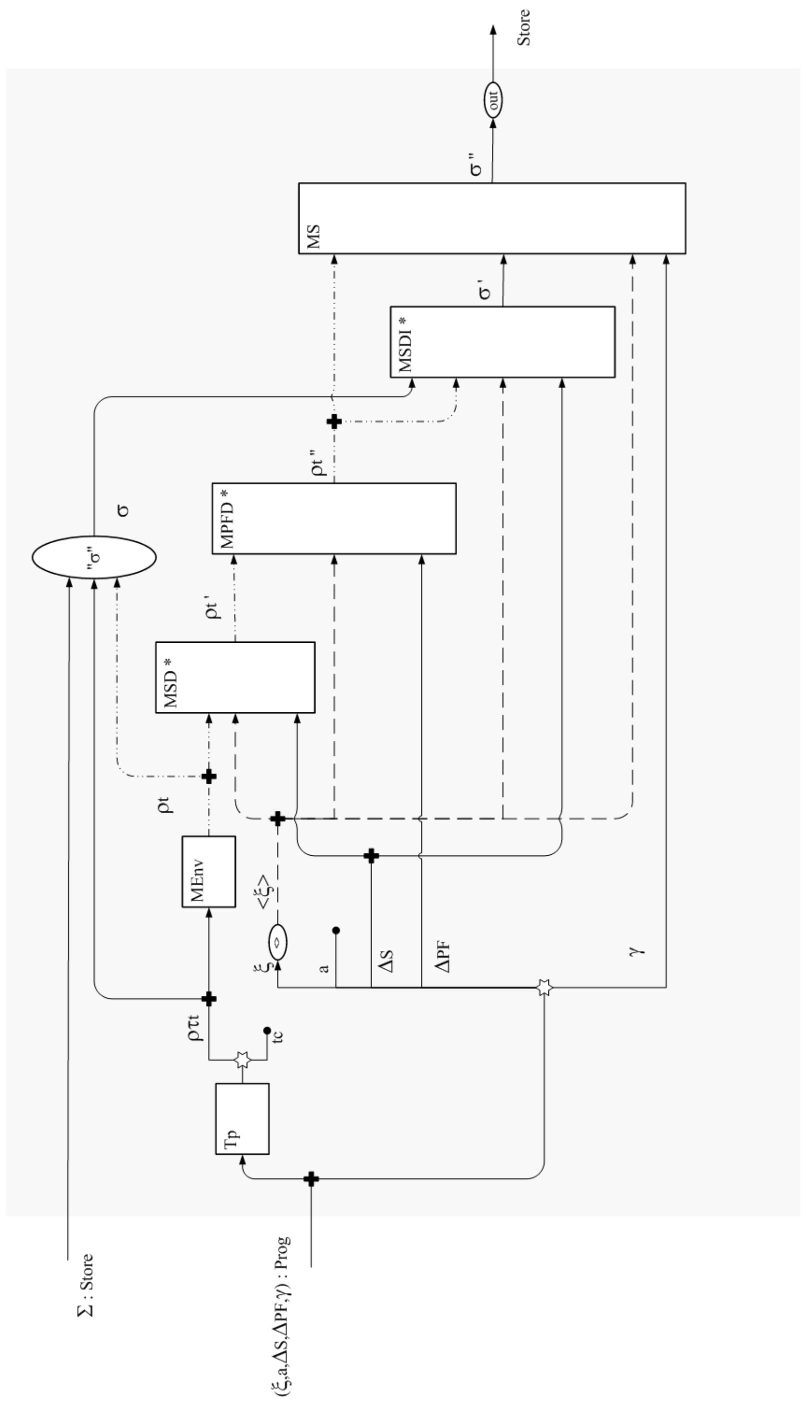

Figure D.2 Z diagram: the structure of the DeCCo Pasp *program* meaning function [Stepney & Nabney 2003, section I.14.4].