Z Unification tools in Generic Formaliser

Eerke Boiten*
Computing Laboratory, University of Kent,
Canterbury, CT2 7NF, U.K.
(Phone: +44 1227 827553,
Email: E.A.Boiten@ukc.ac.uk)

April 1997

Abstract

We describe some prototype tools for performing unification (i.e. deriving the least common refinement) of simple Z specifications. The techniques used are those described in earlier research papers on viewpoint specification in Z; the tools have been implemented in Generic Formaliser (a product of Logica UK Limited).

1 Problem definition

In earlier research papers [BDBS96, BDBS97] we described techniques for constructing a least common refinement, also called *unification*, of two Z [Spi89] specifications in the states-and-operations style. These techniques are mostly of a syntactic nature, so it was clear that they could in principle be automated. We have developed some prototype tools for this purpose.

For an overview of Z unification (and its purpose in viewpoint specification), see our papers [BDBS96, BDBS97], which are also available from the project's WWW site[1]. This note describes only what the prototype tools do – not why they do it. For a quick impression of the tools, the reader is advised to have a glance at the examples in sections 3 and 4. The next section describes the system used for implementing them – Logica UK's *Generic Formaliser*. The remaining sections deal with the tools themselves. A final section lists some conclusions, including ideas on extending these prototypes.

2 Generic Formaliser

Generic Formaliser is a product of Logica UK Limited[2]. It was developed as a generalisation of a tool for Z written by Logica, called (Z Specific) Formaliser. We chose Generic Formaliser as our development framework, since it includes a complete Z grammar that we can modify.

Generic Formaliser is essentially an environment for generating syntax-directed editors. It runs on top of Smalltalk/V for Windows, i.e. only on PCs. Generic Formaliser is actually a syntax-directed editor itself, for the grammar which is appropriately called Grammar Grammar. Grammars (and the production rules in grammars) consist of three parts: an abstract syntax,

*This work was funded by the U.K. Engineering and Physical Sciences Research Council under grant number GR/K13035.

[1] `http://alethea.ukc.ac.uk/Dept/Computing/Research/NDS/consistency`

[2] See `http://public.logica.com/formaliser/`.

various concrete syntaxes (called *unparsings*), and rules for attributes. Functions for computing and checking the attributes are specified in a Z-like syntax, which Generic Formaliser translates into calls of Smalltalk methods. These methods would in general have to be written by grammar writers and added to the Smalltalk environment.

2.1 Implementation techniques

Our purpose was not in the first place to generate syntax directed editors for Z viewpoint specification, thus any use of Generic Formaliser we made was going to be non-standard. What we needed to do was more an issue of syntactic manipulation: transduction or translation of syntax trees. On first thought, the attribute grammar aspect seemed most appropriate for specifying this in. Attribute grammars have the power of a programming language, and an attribute grammar for Z was already provided. However, the final result of manipulations was going to be some representation of a parse tree – but since an editor does not create parse trees for anything but its input grammar, it would have to be returned either as an attribute, or as an unparsing. Having parse trees as attributes would have been possible, but it would have implied the addition of a lot of Smalltalk machinery for actually generating these trees. To reduce the amount of work, we chose for the manipulated tree to appear as a textual (LaTeX) representation in an alternative unparsing (concrete syntax) of the root of the input tree. The grammars for our small tools are based on the Z grammar that comes with Generic Formaliser (Z2), but they do not utilise any of the type checking done through attributes in the Z2 grammar. This is because the Z2 grammar uses Smalltalk shortcuts for delivering attribute values from the root of the parse tree to the leaves, which we would have to recode for our modified grammars.

The use of unparsings as the transduction mechanism has some implications for the (formal) power of what transductions can be described. First, most of the desirable non-context free features of the input grammar could not be described in this way. Thus, part of the functionality of our tools is of the "garbage-in, garbage-out" type, because certain undesirable inputs have not been excluded. One aspect that *is* modelled using unparsings is equality between "subtrees", for example to ensure that the state which is modified by an operation is the state defined in the previous schema. The technique for doing this is to represent such a tree only once in the abstract syntax, and make the unparsings of any further occurrences retrieve this. This has one odd side effect in the generated editors, namely that the user sometimes ends up editing what is conceptually the "copy" of the tree, even if (s)he started out editing the original[3].

2.2 Overview of generated tools

The ideal would of course be to have one single unification tool for the entire Z specification language. However, our theoretical research indicates that the essential aspect of this (in the traditional style of using Z) would be the handling of states and operations. Most other syntactical entities would be included in the unification without change, or at most using simple conjunctions. Thus, our tools deal with state schemas and operation schemas only.

Furthermore, we decided to only write the essential parts of a unifier for those constructs. A full unifying tool would have the following tasks:

1. Collect the state schemas of each viewpoint; compare these lists or use some other criterion to decide which state spaces should be matched up between the viewpoints; ensure that a correspondence is defined for each matching pair.

2. For each such pair of states, *generate the unified state space using the appropriate correspondence.* ("State unification".)

[3]This has been reported, and may be changed in a future Formaliser release.

3. For each modified state space, *replace all the operations defined on it by their adaptations to the unified state space.* ("Operation adaptation".)

4. For each pair of operations on the same (unified) state space that represent the same operation, *generate their operation unification* to replace both operations.

The italicised parts have actually been implemented in tools – clearly the remaining aspects are of a purely book-keeping nature, and their implementation in any programming language would pose no interesting problems. The restriction to these essential steps was also made for very practical reasons: it keeps the grammars relatively small, and allows the use of an (in principle) restricted technique like unparsing. (To put it differently: grammar transformation through unparsings is *not* a full programming language.) The ideal solution for this would not be a single syntax tree transformation tool in any case – it might be a series of such transformations with additional control defined over it.

The next two sections describe the tools that have been implemented. Each consists basically of a grammar with three different unparsings. The first two, *Base* and *LaTeX*, have been taken directly or modified from the Z2 grammar. *Base* provides a nice graphical display of schemas which is convenient for entering specifications; *LaTeX* provides the LaTeX (oz.sty) representation of the input. The third unparsing, called *LaTeXOut*, provides the "output" of the tree transformation. Based on this, it would not be too hard to also generate a graphical version of the output.

3 State unification (and operation adaptation)

In this section we will illustrate the operation of the state unification and operation adaptation tool by showing relevant parts of the grammar for it, and by giving an example.

3.1 Grammar

```
Formaliser grammar for UnifierPhase1and2
Version: Based on Z2; version of March 1997 as described in document
Start symbol: Specifications
Schemes: Base LaTeX LaTeXOut
```

The header of the grammar. The "Schemes" declaration tells us that for each grammar construct, three unparsings exist, viz. the ones we described earlier. We have not listed every unparsing of every rule below. The grammar is called "UnifierPhase1and2" because it covers both state unification and operation adaptation.

```
Specifications :=  <Specification>1
```

A restriction of Formaliser grammars is that the rule for the start symbol should always be a "List1" rule, i.e. it should generate a non-empty list of descendants. We are in principle only interested in one "specification".

```
Specification :=  Viewpoint Viewpoint Correspondence
Unparse scheme Base is 'State unification & adaptation@n@n@1@n@2@nCorrespondenc
e@n@3'
Unparse scheme LaTeX is '@1@2@3
```

A specification consists of two viewpoints (see below for what a viewpoint is in this context) and a correspondence linking the state schemes of these viewpoints. This also illustrates unparsing schemes. The *LaTeX* unparsing is simply the sequence of the *LaTeX* unparsings of the three sons (`@1`, `@2`, `@3`). The *Base* unparsing contains a little syntactic sugar and a few blank lines (`@n`). Most of the actual work in the grammar is done in the *LaTeXOut* unparsing of Specification, as can be seen below. This generates the definitions of types with bottoms, the unified state space, and a consistency condition which needs to hold. Strings like `@ciccic1` [4] refer to *LaTeXOut* unparsings (which at that level are no different from the *LaTeX* unparsings) of nodes deep down in the tree.

```
Unparse scheme LaTeXOut is '\begin{zed}@n@s+\\\t1 +@cccic2_{\bot} @::= bot@ccci
c2 | just@cccic2\lang @cccic2 \rang@s------@n\end{zed}@n\begin{zed}@n@s+\\\t1
......\end{zed}'
```

The dots actually stand for 10 more lines of this kind...

```
Viewpoint :=  SchemaBox Operations
Unparse scheme Base is 'Viewpoint@n@1@n@2'
Unparse scheme LaTeX is '@1@2'
Unparse scheme LaTeXOut is '@1@2'

Operations :=  <OperationBox>
Unparse scheme LaTeX is '@0@d@e'
Unparse scheme LaTeXOut is '@0@d@e'
```

A viewpoint, in the context of this grammar, consists of a state space (using the Z2 grammar concept SchemaBox) and a (possibly empty) sequence of operations. Again the *Base* (on-screen) representation contains a little syntactic sugar. `@0@d@e` is the unparsing scheme syntax for list nodes which simply lists them all without separators or a special representation for the empty list.

```
OperationBox :=  SchemaNameFormals OptDeclPart AxiomPart DeltaXi
Unparse scheme LaTeX is '\begin{schema}{@1}@n@s+\\+@4@ppc1@2@3@s--@n\end{schema
}@n'
Unparse scheme LaTeXOut is '\begin{schema}{Ad@1}@n@s+\\+@4@pppcc1Un@pppcic1@2\w
here @ppccic1@ppccic1 \in \ran just@ppccic2@n @ppccic1@ppccic1'' \in \ran just@
ppccic2@n@s-- \LET\M @ppccic1 == just@ppccic2\inv @ppccic1@ppccic1;@k @ppccic1'
' == just@ppccic2\inv @ppccic1@ppccic1''  \dot  @k@3\O@n\end{schema}@n'

DeltaXi :=  Delta | Xi
```

The definition of OperationBox is very similar to that of SchemaBox in the Z2 grammar, but this is specialised to be an operation on the state just defined. That state is not part of the abstract syntax, but it is listed in every unparsing (`@ppc1`). All that needs to be said additionally is whether this state is changed (Delta) or only used (Xi). The *LaTeXOut* unparsing of this does the actual operation adaptation, for example by prefixing the operation name with `Ad`.

```
Correspondence :=  AxiomPart
Unparse scheme LaTeXOut is '\begin{schema}{Corresp}@n@s+\\+@pcc2@n @pcic2@n\whe
re   @1@s--@n\end{schema}@n'
Unparse scheme LaTeX is '\begin{schema}{Corresp}@n@s+\\+@pcc2@n@pcic2@n@1@s--@n
\end{schema}@n'
```

The correspondence is represented as a schema in all unparsings. In the abstract syntax it is just the AxiomPart of that schema - the declaration part is made up of the declarations of the two viewpoint state schemas (`@pcc2` and `@pcic2`).

[4] `c` is first child, `ci` is second child node; since `1` also refers to a child, this is 5 levels deep in the tree.

3.2 Example

This is the *LaTeX* unparsing of a document presented to this tool. Nothing which occurs twice in this input has had to be entered twice, as explained above with the grammar rule for Correspondence. The first viewpoint has one operation, the second has none.

$State1$

$x : S$

$x + 17 \leq 31$

$Op1$

$\Delta State1;$
$y? : S$

$x = x' - y?$

$State2$

$y : T$

$wellchosen\ y$

$Corresp$

$x : S$
$y : T$

$x\ loves\ y$

The following output (i.e., *LaTeXOut* unparsing) results from this:

$S_\perp ::= botS \mid justS\langle\langle S\rangle\rangle$

$T_\perp ::= botT \mid justT\langle\langle T\rangle\rangle$

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{Corresp}} \\
x : S \\
y : T \\
\hline
x \textit{ loves } y
\end{array}$$

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{State1UnState2}} \\
xx : S_\perp \\
yy : T_\perp \\
\hline
(\exists x : S;\ y : T \bullet \\
\quad xx = justS\ x \land yy = justT\ y \land Corresp \\
\quad \lor yy = botT \land \exists x : S \bullet (xx = justS\ x \land (\forall y : T \bullet \neg Corresp)) \\
\quad \lor xx = botS \land \exists y : T \bullet (yy = justT\ y \land (\forall x : S \bullet \neg Corresp))) \\
\forall x : S \bullet xx = justS\ x \Rightarrow x \\
\qquad\qquad\qquad\qquad +17 \leq 31 \\
\forall y : T \bullet yy = justT\ y \Rightarrow wellchosen\ y
\end{array}$$

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{AdOp1}} \\
\Delta State1UnState2; \\
y? : S \\
\hline
xx \in \operatorname{ran} justS \\
xx' \in \operatorname{ran} justS \\
\textbf{let } x == justS^{-1} xx;\ x' == justS^{-1} xx' \bullet x = x' - y?
\end{array}$$

Consistency condition:

$x \textit{ loves } y \Rightarrow (x + 17 \leq 31 \Leftrightarrow wellchosen\ y)$

3.3 Restrictions

There are a few restrictions which the grammar does not enforce, but which are necessary if one wants sensible output. Most importantly, any state scheme should have only one variable, whose type should be given by an identifier (rather than a general expression). This is so the lists of variables in the unified state can easily be characterised, and so the type-with-bottom has a sensible name already. Also, the names of the variables should be different between the viewpoint state schemas.[5]

Additionally, the input schemas cannot be checked for the Z static semantics (names being

[5]This restriction is similar to the one usually imposed (but rarely mentioned) for data refinement in Z: for an abstraction schema to make sense, it should be between variables with different names.

defined, type-correctness, etc.). The attribute rules for doing so are identical or almost similar to the corresponding rules from Formaliser's standard Z Grammar, but would need some Smalltalk coding in order to reimplement efficiency shortcuts used in that grammar.

4 Operation unification

This section is structured similarly to the previous one: aspects of the grammar explained, an example, and a short list of restrictions on the input.

4.1 Grammar

```
Formaliser grammar for UnifierPhase3
Version: March 1997, based on Z2, as described in document
Start symbol: Specifications
Schemes: Base LaTeX LaTeXOut

Specifications :=  <Specification>1
Unparse scheme LaTeX is '@0@n@d@e'
Unparse scheme LaTeXOut is '@0@n@d@e'
```

As in the previous section, there are three unparsings: screen, LaTeX input and LaTeX output. Also the top level rule is the same, with "specification" denoting the unit we are actually interested in.

```
Specification :=  FirstOp SecondOp
Unparse scheme Base is 'Operation unification: two representations of@none oper
ation in different viewpoints.@n@nIn viewpoint 1:@n@1@nIn viewpoint 2:@n@2'
Unparse scheme LaTeX is '@1@2'
Unparse scheme LaTeXOut is '@1@2\begin{schema}{UnOp}@s+\\ +@n@c4@c5@c2@ci2\wher
e@n\pre @c1 \Rightarrow @c3@n \pre @ci1 \Rightarrow @ci3@s---@n\end{schema}@n@n
{\bf Consistency condition:}@n\begin{zed}@n \pre @c1 \wedge \pre @ci1 \Rightarr
ow @c3 \wedge @ci3@n\end{zed}@n'
```

A specification this time consists of two operations on the same state space that need to be unified. The output is generated mostly in the *LaTeXOut* unparsing of this rule, including a consistency condition that needs to be satisfied for the unification to be a least common refinement of the input operations. The rules below demonstrate why different nonterminals need to be used for the first and second operation.

```
FirstOp :=  SchemaNameFormals OptDeclPart AxiomPart DeltaXi SchemaName
Unparse scheme LaTeX is '\begin{schema}{@1}@n@s+\\+ @4@5 @2 @3@s--@n\end{schema
}@n'
Unparse scheme LaTeXOut is '\begin{schema}{@1}@n@s+\\+ @4@5  @2 @n@b@b\where@n@
b@b@3@s--@n\end{schema}@n'

SecondOp :=  SchemaNameFormals OptDeclPart AxiomPart
Unparse scheme LaTeX is '\begin{schema}{@1}@n@s+\\+@pc4@pc5 @2 @3@s--@n\end{sch
ema}@n'
Unparse scheme LaTeXOut is '\begin{schema}{@1}@n@s+\\+ @pc4@pc5  @2  @n@b@b\whe
re@n@b@b@3@s--@n\end{schema}@n'
```

```
DeltaXi :=  Delta | Xi
```

Both of these are variations on the standard SchemaBox. However, we needed to ensure here that both operate on a common state. This is done by including DeltaXi and SchemaName in the first operation (only); all unparsings of the second operation inherit those (`@pc4` and `@pc5`) from those of the first operation.

4.2 Example

This is the *LaTeX* unparsing of a document presented to this tool.

$Op1$

$\Delta State$

$true$

$Op2$

$\Delta State;$

$x? : \{1\}$

$false$

The following output (i.e., *LaTeXOut* unparsing) results from this:

$Op1$

$\Delta State$

$true$

$Op2$

$\Delta State;$
$x? : 1$

$false$

$UnOp$

$\Delta State$
$;$
$x? : 1$

$\mathrm{pre}\ Op1 \Rightarrow true$
$\mathrm{pre}\ Op2 \Rightarrow false$

Consistency condition:

$$\mathrm{pre}\ Op1 \land \mathrm{pre}\ Op2 \Rightarrow true \land false$$

4.3 Restrictions

According to the strict Z refinement rules [Spi89], the OptDeclParts of the two operations should have the same (i.e., both names and types) lists of inputs and outputs for both operations. This is not enforced by this grammar, also because we feel this might be too restrictive for viewpoint specification.

Like for the state unification tool, static semantics of the input specification is not checked.

5 Concluding remarks

These prototype tools have shown that it is indeed possible to support viewpoint specification in Z with tools for combining the viewpoints. Also, they are able to generate the consistency conditions that need to hold, which can then serve as inputs to theorem provers, for example Z in Isabelle [KB95]. The generated output (LaTeX for oz.sty) can (possibly after minor modifications) be fed into various systems for analysis of Z specifications.

It has taken a relatively long time to produce these prototype tools. This is partially due to the fact that the functionality of Generic Formaliser was being extended throughout this period, to match its specifications and to match our requests for extra functionality. In the end, many features of Generic Formaliser were left unused, unfortunately. One could imagine next versions of these tools making use of attributes for checking static semantics (unused now because of shortcuts mentioned before). Also, attributes containing strings or parse trees would allow a more modular (and slightly more flexible) generation of unifications. Functions for manipulating parse trees are already available with Generic Formaliser; one would have to add a few more of these, and/or functions for manipulating string attributes (this implies writing Smalltalk methods). Using the existing feature of attributes used in unparsings, such strings could easily be combined into unparsings, which would not be as complicated as the unparsing strings we now use.

Acknowledgements

Susan Stepney of Logica Cambridge was extremely helpful with quick responses to questions on Formaliser, and quick fixing of bugs and extensions of Formaliser functionality. Despite the sometimes frustrating results of using Generic Formaliser in a way which was far from its original purpose, it was a pleasure to cooperate on this with her.

More information on the project "Viewpoint Consistency in Open Distributed Processing", including papers on viewpoint unification in Z, may be found on:
`http://alethea.ukc.ac.uk/Dept/Computing/Research/NDS/consistency`.

References

[BDBS96] E. Boiten, J. Derrick, H. Bowman, and M. Steen. Consistency and refinement for partial specification in Z. In M.-C. Gaudel and J. Woodcock, editors, *FME'96: Industrial Benefit of Formal Methods, Third International Symposium of Formal Methods Europe*, volume 1051 of *Lecture Notes in Computer Science*, pages 287–306. Springer-Verlag, March 1996.

[BDBS97] E.A. Boiten, J. Derrick, H. Bowman, and M. Steen. Constructive consistency checking for partial specification in Z. Extended version of [BDBS96]; submitted for publication, 1997.

[KB95] I. Kraan and P. Baumann. Implementing Z in Isabelle. In J. P. Bowen and M. G. Hinchey, editors, *ZUM'95: The Z Formal Specification Notation, 9th International Conference of Z Users, Limerick, Ireland, September 7-9, 1995, Proceedings*, volume 967 of *Lecture Notes in Computer Science*, pages 355–373. Springer-Verlag, 1995.

[Spi89] J. M. Spivey. *The Z notation: A reference manual.* Prentice Hall, 1989.