Composing grammar transformations to construct a specification of a parser

Luke Wildman *Ian Hayes*

Software Verification Research Centre

Department of Computer Science
University of Queensland
Brisbane, 4072
email: wild@cs.uq.oz.au

Abstract

As part of a project with the aim of scaling up formal methods, we have developed a library construct for the specification language Z. This paper reports on the result of using libraries to structure a specification of a relatively complicated parser for a language-based editor. The parser is complicated by the need to cope with multiple languages as well as tolerate errors in the input.

Our goal in producing the specification of the parser has been to separate each of the major concepts on which the specification is based (eg, multiple languages and error-tolerance) into a separate library.

To achieve the separation of concerns we have applied the novel technique of specifying each of the major concepts of the parser as grammar transformations. The full parser can then be specified by composing the separate transformations to give a grammar incorporating all the desired features.

1 Introduction

In working towards a suitable framework for writing large specifications we have developed a library construct [2] that allows a Z [8, 3] specification to be presented in a structured form. In this paper we examine the application of libraries to the specification of an extended parser for a structured editor.

The editor A generic language-based editor [1] has been developed to provide support for software development. Two major features of the editor are the support of multiple languages in a single document (eg, a programming language plus a specification language) and the incorporation of an incremental parser [5, 4] that is tolerant of errors introduced into a document via edit operations in the course of modifying the document.

The parser For the specification of a parser for a simple language-based editor it is appropriate to use the well-known concepts of formal language theory. This theory allows one to relate a grammar for the language being edited to strings of symbols (language recognition) and to define parse trees for those strings (parsing). The language-based editor incorporates three major extensions:

- it is tolerant of errors in the input string during the editing process;
- it supports multiple languages within a single document; and
- it supports *contexts* which correspond to the block or module structure of the language being edited.

There are two approaches that we have considered to incorporate each of these extensions into the parser specification:

1. develop a specialised formal language theory to support the extension directly, or
2. reuse the basic formal language theory by encoding the extensions in the grammars.

As an example of the first approach, the special theory we developed for an error-tolerant parser used a more complex parse tree representation with special nodes to cope with both missing and extraneous symbols in the input string, and a more complex parsing relationship between the grammar and the extended parse tree. The second approach is to transform the grammar for the language into an extended grammar with additional symbols and productions that are used to encode the error tolerance.

Having considered both these approaches, we prefer the grammar transformation approach because

- it is simpler to define the transformation of the grammar than to define a specialised language theory,
- the basic formal language theory developed for a simple grammar can be reused directly on the extended grammar, and
- most importantly, as our application extends the parser in three different ways, using the grammar transformation approach allows us to apply three separate transformations to our grammars to achieve the desired net result. This is only possible because all the extensions share the same basic formal language theory.

With the specialised theory approach life is more complicated. Although it is reasonably straightforward to develop a specialised formal language theory for each of the three extensions, there is no simple way to combine the extended theories. Instead one is forced to develop a monolithic, very complicated, language theory that incorporates all three extensions. Discarding this approach in favour of grammar transformations allows a clearer separation of the multiple extensions and, overall, a simpler specification.

A brief introduction to the library mechanism is presented in Section 2. Some examples of the transformations specified in each library are given in Section 3. In Section 4 we discuss the composition of the transformations to form the specification, and in Section 5 we justify the approach taken with some examples of the improvements made over previous specifications.

2 Libraries: an introduction

The addition of a library facility to Z was motivated by a desire to support a modular approach to specification. The objectives were to be able to build a specification from components each describing a single aspect of the specification, as well as to allow for component reuse. To support reuse some form of parameterisation was desirable.

In Z one considers a specification to be a document; therefore it is quite natural to consider a library to be a new section (or subsection), of such a document.

At its simplest a library is a named collection of type and variable definitions. An example is the *Grammar_types* library (Section 3.1). A library is similar to a specification document or a Z chapter [6] except that the definitions of a Z chapter are visible to all of the following chapters of the specification. Normally, libraries must be explicitly instantiated to gain access to the definitions contained within. However, *Global* libraries (Section 2.2) are similar to the Z chapter mechanism in that their definitions are visible to the entire specification.

A normal library heading contains its name and any formal type parameters (see Section 2.1). The body of a library contains definitions (of types and variables) which may be dependent on the library's type parameters. One instantiates a library thus

instantiate *library*1

The effect is to provide access to the definitions contained therein as if they had been written at the point of instantiation. The scope of the instantiation extends to the end of the library or document in which it is instantiated.

2.1 Generic Libraries

Libraries may have parameters. The parameters allowed are unstructured sets similar to those currently allowed in Z for generic definitions. This allows one to group together a collection of related definitions *generic as a whole* in the parameter sets. That is, when the library is instantiated, all of the component definitions are instantiated with the same parameter sets. An example is the library *Multiple_grammars* (Section 3.2). When it is instantiated, its generic sets (L and SY) must be provided. The instantiation of the library provides all the definitions within the library, each instantiated with the actual parameter sets. This separates the instantiation of a library from the use of that instantiation's definitions and therefore removes the need for the uniqueness constraint put on Z generic definitions [7, p.85]; once an instantiation is made, each use of a definition from that instantiation refers to the same definition.

Note that this does not preclude the use of the usual generic Z definitions where each definition is individually dependent on a generic set. Libraries provide an additional mechanism for introducing a group of related definitions generic as a whole in the parameter sets.

2.2 Global libraries

It is impractical to explicitly instantiate the definitions for the standard Z toolkit in every library written. We use a set of *global libraries* which are

given global scope over the entire specification. The global libraries include the Z toolkit but may also be extended with any non-generic library. The set of global libraries for a given specification may be extended by using the keywords **Global library** to introduce a new library rather than just **Library**. The library *Grammar_types* (Section 3.1) is an example of a global library.

2.3 Schemas vs libraries

A library is a generalisation of a Z schema. The main differences between them are that

- a library allows new types to be created in its body, whereas this can not be done in a schema, and
- a schema can be used as a type, whereas a library can not.

The library extension to Z is discussed in greater detail in [2].

A typechecker for the library extension to Z has been developed by extending the *hippo* typechecker for Z developed by Sufrin *et al* [9]. The specification presented in this paper has been mechanically checked with the typechecker.

3 Grammar transformations

In this section we introduce some of the libraries that define the grammar transformations that are used within the specification of the parser for the language-based editor. Because of space limitations we cannot give all the libraries used in the specification of the parser. In particular we do not cover the transformations that support contexts within the editor (see [10] for more detail).

The global library *Grammar_types* (Section 3.1) introduces the basic grammar type used in the specification. The library *Multiple_grammars* (Section 3.2) specifies a transformation from a set of monolingual grammars into a single multi-lingual grammar. The library *Error_grammars* (Section 3.3) specifies a transformation from a single normal grammar into the corresponding error-tolerant grammar.

3.1 Global library: Grammar_types

The set of productions is represented by a relation between a symbol on the left side of a production and a sequence of symbols on the right side that it may produce. The type *Productions* is generic in the set of symbols SY.

$$Productions[SY] == SY \leftrightarrow \operatorname{seq} SY$$

A context-free grammar consists of sets of productions, P, terminal symbols, T, nonterminal symbols, NT, and starter symbols, SS. The nonterminal symbols may have productions defined for them and may be used in the productions of other nonterminals and so may appear on both left and right sides. Terminal symbols do not produce any other symbols and therefore may only appear on the right side of a production. The starter symbols are special nonterminals that may be used as the root of a derivation sequence. Usually a grammar has only one starter symbol however the editor allows multiple starter symbols so we have accommodated this generalisation.

$$\begin{array}{|l} Grammar[SY] \\ \hline P : Productions[SY] \\ SS, NT, T : \mathbb{P}\, SY \\ \hline SS \subseteq NT \land T \cap NT = \{\} \\ \operatorname{dom} P \subseteq NT \\ \forall\, rhs : \operatorname{ran} P \bullet \operatorname{ran} rhs \subseteq NT \cup T \\ \hline \end{array}$$

In the full specification [10] there are several additional libraries that develop formal language and parsing theory. The *Derivations* library further develops formal language theory. It describes how one sequence of symbols may directly or indirectly *derive* another sequence of symbols, as well as concepts like *starter sets* and *languages*. A general *Tree* library provides useful definitions to do with multiway trees and a more specific *ParseTree* library describes how a parse tree is related to a grammar. These libraries have been omitted here as they are not necessary for the presentation of the following grammar transformations.

3.2 Library: Multiple_grammars[SY,L]

The parser has the capability to process many different languages simultaneously. We deal with this by combining all the language grammars into a single grammar. This library is parameterised by the base symbol set for all the grammars, SY, and by the set of language identifiers, L. The set of grammars to be merged is of type *MultiGram*, which maps a language to the grammar for that language.

$$MultiGram == L \pfun Grammar[SY]$$

The grammars may share common symbols and we have to be careful to maintain the distinction between them. We also wish to be able to determine

the source language of each symbol. We construct a new symbol type, *MLS*, which incorporates both the language and the original symbol.

$$MLS == L \times SY$$

By constructing our merged multi-lingual grammar with these language-symbol pairs as symbols we ensure that the symbols used in the grammar for one language do not overlap with the symbols of the grammars for any other language. The function *addl* converts a symbol to such a language-symbol pair.

$$\begin{array}{|l} addl : L \to (SY \to MLS) \\ \hline \forall\, l : L;\ s : SY \bullet addl(l)(s) = (l, s) \end{array}$$

It is defined in curried form so that we may transform a set of symbols *ps* to a set of language-symbol pairs of language *l* by taking the relational image of *ps* through the function $addl(l)$, i.e., $addl(l)(\!|\ ps\ |\!)$.

The function *combine* transforms a set of grammars to produce a single grammar of *MLS* symbol type. The terminal symbols of the combined grammar are the union over all the languages of each language's terminal symbols, each of which has been augmented with its language identifier. The nonterminals and starters are treated similarly. The same treatment is also applied to the productions: both the nonterminal (n) on the left and all the symbols in the sequence of symbols (rs) on the right side of a production must be augmented with the language identifier.

As an example consider the multiple grammar *mg* which contains the grammars for two languages based on the symbol set

$$SY ::= a \mid b \mid c \mid d.$$

The construct $(\mu\, Grammar[SY] \mid P)$ creates a unique element of the schema type *Grammar* with fields satisfying the predicate *P*.

$$\begin{array}{l} mg == \\ \quad \{l1 \mapsto (\mu\, Grammar[SY] \mid \\ \qquad T = \{a, b\} \land NT = \{c\} \land \\ \qquad SS = \{c\} \land P = \{c \mapsto \langle a\rangle, c \mapsto \langle a, b\rangle\}), \\ \quad\ l2 \mapsto (\mu\, Grammar[SY] \mid \\ \qquad T = \{a, c\} \land NT = \{d\} \land \\ \qquad SS = \{d\} \land P = \{d \mapsto \langle a, d\rangle, d \mapsto \langle c\rangle\})\} \end{array}$$

The result of the combination is

$$\begin{array}{l} combine(mg) = (\mu\, Grammar[MLS] \mid \\ \quad T = \{(l1, a), (l1, b), (l2, a), (l2, c)\} \land \\ \quad NT = \{(l1, c), (l2, d)\} \land \\ \quad SS = \{(l1, c), (l2, d)\} \land \\ \quad P = \{(l1, c) \mapsto \langle (l1, a)\rangle, \\ \qquad (l1, c) \mapsto \langle (l1, a), (l1, b)\rangle, \\ \qquad (l2, d) \mapsto \langle (l2, a), (l2, d)\rangle, \\ \qquad (l2, d) \mapsto \langle (l2, c)\rangle\}) \end{array}$$

The specification of *combine* follows.

$$\begin{array}{|l} combine : MultiGram \to Grammar[MLS] \\ \hline \forall\, mg : MultiGram \bullet \\ \textbf{let}\ lg == \mathrm{dom}\ mg \bullet \\ combine(mg) = (\mu\, Grammar[MLS] \mid \\ \quad T = \bigcup\{l : lg \bullet addl(l)(\!|\ mg(l).T\ |\!)\} \land \\ \quad NT = \bigcup\{l : lg \bullet addl(l)(\!|\ mg(l).NT\ |\!)\} \land \\ \quad SS = \bigcup\{l : lg \bullet addl(l)(\!|\ mg(l).SS\ |\!)\} \land \\ \quad P = \bigcup\{l : lg \bullet \{n : SY;\ rs : \mathrm{seq}\, SY \mid \\ \qquad (n, rs) \in mg(l).P \bullet \\ \qquad addl(l)(n) \mapsto map(addl(l))(rs)\}\}) \end{array}$$

The function *map* applies a given function to each element of a sequence yielding the sequence of results. In this case it yields a sequence of language-symbol pairs.

3.3 Library: Error_grammars[SY]

In this library we specify a function which transforms a normal grammar into an error-tolerant grammar.

A feature of the editor is the ability of the parser to tolerate errors in the document at edit points. The editor copes with both missing and extraneous symbols in the input while editing the parse tree. It copes with missing symbols by leaving a *hole* symbol in the vacant position and with extraneous symbols by placing them in an *error node* which is grafted into the parse tree at the position where the errors occurred in the input.

Our approach is to transform a grammar for a language into an error-tolerant grammar, which includes productions involving hole and error symbols. First we extend the symbol set with the error symbols.

The type parameter to this library, *SY*, represents the normal symbols in the unextended grammar. This set needs to be extended with the error symbol *E*, and the hole symbol *H*. As part of the transformation, the terminal symbols of the original grammar are promoted so that they may produce a hole and an error. To accommodate this we also need to extend the symbol set with

a set of replacement terminal symbols: $ET\langle\langle SY\rangle\rangle$. The following disjoint union defines the complete extended symbol set.

$$ES ::= ESnorm\langle\langle SY\rangle\rangle \mid ET\langle\langle SY\rangle\rangle \mid E \mid H$$

The normal symbols are embedded in the alternative *ESnorm*.

As an example, if the parameter set is $SY = \{a, b, c\}$, then the extended set is

$$ES = \{ESnorm(a), ESNorm(b), ESNorm(c), ET(a), ET(b), ET(c), E, H\}$$

That is the elements formed by the constructor functions *ESnorm* and *ET* and the constants *E* and *H*.

We specify a transformation function *errgram* which takes as input a grammar which is not error tolerant and gives as output that grammar with its symbols and productions extended to make it error tolerant. To aid in constructing the transformation we first introduce a number of auxiliary functions that are to be used in its definition.

In order to allow extraneous material to be stored wherever it occurs, error symbols are placed on the end of every production. This means that every parse tree will have an error tree as its last child. If an error tree has no subtrees then there is no error.

The function *adderrors* adds error symbols onto the end of productions and promotes the normal symbols in the productions to the extended symbol set *ES* by using the embedding function *ESnorm*. The *error* symbol must be added to the nonterminal symbols.

$$\begin{array}{|l} adderrors : \\ \quad Productions[SY] \to Productions[ES] \\ \hline \forall\, P : Productions[SY] \bullet \\ \quad adderrors(P) = \{s : SY;\ ss : \operatorname{seq} SY \mid \\ \qquad s \mapsto ss \in P \bullet \\ \qquad\quad ESnorm(s) \mapsto (map(ESnorm)(ss) \frown \langle E\rangle)\} \end{array}$$

Error symbols can derive any sequence of symbols including the empty sequence. The empty case corresponds to there being no error. We define productions for the error symbol which generate any sequence of non-error symbols.

$$\begin{array}{|l} errorprod : Productions[ES] \\ \hline errorprod = \{os : \operatorname{seq}(ES \setminus \{E\}) \bullet E \mapsto os\} \end{array}$$

The *holeprod* productions are added so that every symbol may optionally be replaced by a *hole* symbol. In addition, we add an error node to the end of the right side of the production. This allows, for example, an incorrect symbol to be treated by leaving a hole and then placing the symbol in the following error node subtree. The *hole* symbol must be added to the set of terminal symbols to facilitate this.

$$\begin{array}{|l} holeprod : Productions[ES] \\ \hline holeprod = \{s : SY \bullet ESnorm(s) \mapsto \langle H, E\rangle\} \end{array}$$

The *ETprod* productions are added so that each original terminal symbol produces a new *ET* terminal symbol as an alternative to a *hole* symbol. All the original terminal symbols become nonterminal symbols and the new ET terminal symbols replace the original terminal symbols. As before an error node is also added to the right side of the productions. The set of terminal symbols is a parameter to *ETprod*.

$$\begin{array}{|l} ETprod : \mathbb{P}\, SY \to Productions[ES] \\ \hline \forall\, terminals : \mathbb{P}\, SY \bullet \\ \quad ETprod(terminals) = \{t : terminals \bullet \\ \qquad ESnorm(t) \mapsto \langle ET(t), E\rangle\} \end{array}$$

The grammar transformation We are now in a position to describe the complete transformation of a grammar, *g*, to an error-tolerant grammar. The terminal symbols of the new grammar are the hole symbol plus the new *ET* symbols introduced to replace the old terminals. The new non-terminals are the error symbol plus the promoted old non-terminals and terminals. The new starters symbols are the promoted old starters. All of the original productions must be both promoted to the new symbol type and extended with an *error* symbol to allow an error production to be used when needed. The new productions consist of the promoted old productions, $adderrors(g.P)$, plus productions allowing every symbol to produce a hole, *holeprod*, plus productions allowing the promoted old terminal symbols to produce the new terminal symbols, $ETprod(g.T)$, plus productions allowing the error symbol to produce any string of symbols, *errorprod*.

$$\begin{array}{|l} errgram : Grammar[SY] \to Grammar[ES] \\ \hline \forall\, g : Grammar[SY] \bullet \\ errgram(g) = (\mu\, Grammar[ES] \mid \\ \quad T = ET(\!|\ g.T\ |\!) \cup \{H\} \\ \quad NT = ESnorm(\!|\ g.NT \cup g.T\ |\!) \cup \{E\} \\ \quad SS = ESnorm(\!|\ g.SS\ |\!) \\ \quad P = adderrors(g.P) \cup holeprod \cup \\ \qquad ETprod(g.T) \cup errorprod) \end{array}$$

4 Composing the transformations

This section is part of the top-level specification which uses the libraries defined previously to build a multi-lingual, error-tolerant grammar.

The global library *Grammar_types* contains the generic definition of *Grammar* which is used by both of the other libraries. A parser is generated from a set of *grammars*, one for each language. The symbol type, SY, and language identifiers, L are given types defined at the top level.

$$| \; grammars : L \nrightarrow Grammar[SY]$$

Our first task is to transform each of these grammars into an error-tolerant grammar. We instantiate the library *Error_grammars* which defines the extended symbol set *ES* and provides the function *errgram* to transform a grammar into an error-tolerant grammar.

instantiate *Error_grammars*$[SY]$

To combine the error-tolerant grammars of a set of languages into a single multi-lingual grammar, we instantiate the library *Multiple_grammars*. The instantiated symbol type is *ES* because the grammars to be combined are error-tolerant and hence based on the extended symbol set. This instantiation provides us with the grammar combining transformation *combine*.

instantiate *Multiple_grammars*$[ES, L]$

The composition The transformation functions defined in the libraries can be composed because they match on their grammar types. The *errgram* transformation is composed with the function *grammars*, which denotes the set of mono-lingual grammars, to form a set of mono-lingual, error-tolerant grammars. The combined multi-lingual, error-tolerant grammar *combinedgram* is defined by applying the *combine* transformation to the result.

$$\begin{array}{|l} combinedgram : Grammar[MLS] \\ \hline combinedgram = \\ \quad combine(errgram \circ grammars) \end{array}$$

The composition could be done in the other order, ie, combine the set of mono-lingual grammars and extend the result with error-tolerance. To do this we must instantiate the libraries *Error_grammars* and *Multiple_grammars* in the reverse order.

instantiate *Multiple_grammars*$[SY, L]$
instantiate *Error_grammars*$[MLS]$

We can then define a combined grammar of the type *ES* introduced by the instantiation of *Error_grammars*$[MLS]$.

$$\begin{array}{|l} combinedgram : Grammar[ES] \\ \hline combinedgram = \\ \quad errgram(combine(grammars)) \end{array}$$

This produces a different result. Instead of having many different error and hole symbols, one of each for each language, it has only one error and one hole symbol, each of which is independent of language. This is not suitable for our application because we want the symbols produced from an error symbol to come from one language only. However, if one wanted to allow error symbols to produce symbols from any language then the alternative combination could be used. That we have the ability to combine the extensions in either way pays credit to the power of a specification constructed using transformations.

5 Conclusion

Grammar transformations The specification presented in this paper is the result of several iterations. The earliest version of the specification (not presented here) used a specialised theory of parsing that incorporated error-tolerance and multiple languages via the use of many special node types within the parse tree. The 'specification' was monolithic, complex and difficult to understand. In fact, it was close to an algorithmic implementation of the parser. Furthermore, there were virtually no components of the specification that could be reused for another purpose; the components were all specialised to the particular application at hand. Without reproducing the original specification here, something we would not want to foist upon our reader, we hope that we have given you an impression of the inadequacies of the original specification. Dissatisfaction with the result of this approach led us to look for better ways of building the specification.

Our first discovery was that we could avoid developing a specialised parsing theory by transforming a set of grammars into a single, multi-lingual, error-tolerant grammar and then using standard parsing theory on the result. That was perhaps the most important discovery. Not only did it allow us to reuse existing parsing theory, it also made it possible to decompose the grammar transformation into separate component transformations corresponding to the different facets of the

parser. That was our second discovery. Only after having made both these discoveries was it possible to modularise the specification into separate coherent components each of which deals with only a single facet of the parser, and each of which may be understood in isolation.

The reason the transformation style works for this application is that each transformation preserves the basic structure of the parameter: a grammar. The input to a transformation is a grammar, or a set of grammars, and the output is also a grammar. Although the grammars may be based on different symbol sets they share the same structure. This makes it possible to compose several such transformations to achieve a complex overall transformation.

The individual transformations can be thought of as building blocks. No transformation depends on the presence of a previous transformation and so the transformations can be performed in any combination. However, each combination gives a different result. This gives us the flexibility to produce a range of different specifications each corresponding to a different combination of the primitive building blocks.

In the full specification, not only are transformations used to add multiple languages and error-tolerance to the parser, an additional transformation is used to add contexts to the parser (see [10] for more detail). Contexts allow the multiple languages to be used together.

Libraries The earlier versions of the specification were presented in standard (flat) Z. As part of our work on a simple modularisation facility for Z [2] we decided to produce a version of the specification using libraries. Our aim in modularising the specification was to split it into a number of libraries, each describing a single aspect of the specification.

In revising the specification we have developed a set of libraries for dealing with grammars. Some libraries are more widely applicable than others. For instance, the library *Grammar_types*, would be useful in any specification requiring language productions. The library *Multiple_grammars* (Section 3.2) is also quite general and could be incorporated into other specifications. The library *Error_grammars* (Section 3.3) is specific to the particular application. However having it as a library helps structure the specification.

Modularising the specification forced us to think about the overall structure of the specification more than we had done in producing the flat Z specification. In the earlier specifications concepts that have now been separated were entangled. For example, the treatments of errors and multilingual grammars are conceptually quite distinct and, given the grammar transformation technique described above, it was not difficult to disentangle them to produce separate libraries that can be used independently.

In this paper we have presented the final product of several iterations of specification. Although the specification is still complex, we feel that this is due to the underlying complexity of the parser itself, rather than the inadequacies of the specification techniques used.

Some valuable lessons relevant to building large specifications in general are the separation of concerns and the construction of coherent components. Each library develops a single concept which makes it easier to follow and reusable in more situations. The reader can concentrate on the single concept in isolation and is more likely to understand its purpose. When the library is used it is only the top-level concept developed within the library that the reader need be aware of, not all the details used to define it. This makes the top-level specification easier to follow.

As an exercise to test the capabilities of a simple modularisation facility for Z this case study has been a success. The full capabilities of the library mechanism are not used in this paper because we only use one instantiation of each library, however, the example does demonstrate the use of libraries for structuring a specification. The simple library construct has aided us to produce a better-structured specification and has been flexible enough that it has not forced artificial constraints on the structure.

Acknowledgments

Luke Wildman was supported by an Australian Postgraduate Research Award. We would like to acknowledge our collaboration with Jim Welsh and David Carrington on an Australian Research Council supported grant entitled *Modularity in the Derivation of Verified Software* (A48931426).

References

[1] B. Broom, J. Welsh and L. Wildman. UQ2: a multilingual document editor. In *Proceedings of the 5th Australian Software Engineering Conference*, May 1990.

[2] I. Hayes and L. Wildman. Towards Libraries for Z. In John E. Nicholls (editor), *Proceedings*

of the Seventh Annual Z User Meeting, London, December 1992, Workshops in Computing, pages 37–51. Springer-Verlag, December 1993.

[3] I.J. Hayes (editor). *Specification Case Studies*. International Series in Computer Science. Prentice Hall, Hemel Hempstead, Hertfordshire, UK, 2nd edition, 1993.

[4] D. Kiong. *Program Analysis in Language-based Editors*. Ph.D. thesis, University of Queensland, 1987.

[5] D. Kiong and J. Welsh. An incremental parser for language-based editors. In *Proc. 9th Australian Computer Science Conference*, pages 107–118, Canberra, 29-31 January 1986.

[6] I.H. Sørensen. A specification language. In J. Staunstrup (editor), *Program Specification: Proceedings of a Workshop*, Volume 134 of *Lecture Notes in Computer Science*, pages 381–401. Springer-Verlag, 1981.

[7] J.M. Spivey. *Understanding Z: A Specification Language and its Formal Semantics*, Volume 3 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, UK, January 1988.

[8] J.M. Spivey. *The Z Notation: A Reference Manual*. International Series in Computer Science. Prentice Hall, Hemel Hempstead, Hertfordshire, UK, 2nd edition, 1992.

[9] J.M. Spivey and B.A. Sufrin. Type inference in Z. In D. Bjørner, C.A.R. Hoare and H. Langmaack (editors), *VDM and Z – Formal Methods in Software Development*, Volume 428 of *Lecture Notes in Computer Science*, pages 426–438. VDM-Europe, Springer-Verlag, 1990.

[10] L. Wildman and I. Hayes. A specification of a parser for UQ2: an error-tolerant multilingual language-based editor. Department of Computer Science, University of Queensland, 1993.