TOZE - A Graphical Editor for the Object-Z Specification Language with Syntax and Type Checking Capabilities

A Manuscript
Submitted to
the Department of Computer Science
and the Faculty of the
University of Wisconsin-La Crosse
La Crosse, Wisconsin

by

Tim Parker

in Partial Fulfillment of the
Requirements for the Degree of

Master of Software Engineering

December, 2008

TOZE - A Graphical Editor for the Object-Z Specification Language with Syntax and Type Checking Capabilities

By Tim Parker

We recommend acceptance of this manuscript in partial fulfillment of this candidate's requirements for the degree of Master of Software Engineering in Computer Science. The candidate has completed the oral examination requirement of the capstone project for the degree.

_____	_____
Dr. Kasi Periyasamy Examination Committee Chairperson	Date
_____	_____
Dr. Kenny Hunt Examination Committee Member	Date
_____	_____
Dr. Mark Headington Examination Committee Member	Date

ABSTRACT

Parker, Tim, T, "TOZE – A Graphical Editor for the Object-Z Specification Language with Syntax and Type Checking Capabilities", Master of Software Engineering, December 2008, Advisors: Kasi Periyasamy, Ph.D. , Kenny Hunt, Ph.D..

This manuscript describes the development of a tool that allows users to create and edit formal specifications in the Object-Z language using a graphical user interface. The tool enables users to enter Object-Z specifications, validate correctness of the syntax, and check for type inconsistencies. The vision for the editor was to make it work similar to a WYSIWYG word processor, like Word, where the user is able to work with the content as it would appear on a printed page. Unlike other tools for Object-Z, TOZE provides interactive facilities to check syntax and types within the tool without the need to leave the editor and use other applications. Basic file management functionalities such as saving and opening specifications as well as more advanced features such as exporting the specification as a JPEG image or LaTeX document are provided.

ACKNOWLEDGEMENTS

I would like to express my sincere appreciation to my project advisors Kasi Periyasamy, Ph.D. and Kenny Hunt, Ph.D.. Their guidance and knowledge of Z and Object-Z was invaluable during the project. I would also like to acknowledge the CS-742 class of spring 2008 who served as real-world testers for the tool. During the course of the semester, they came up with specifications that pushed the tool to its limits and provided feedback that helped make the tool much more robust. Finally, I would like to thank my wife Amy and our children Tyler and Bailey for their support.

TABLE OF CONTENTS

LIST OF FIGURES

GLOSSARY

Document Markup Language

A language consisting of a set of keywords that annotate text within a document. It is used to control how the document is rendered.

Java

An object-oriented programming language that is platform independent.

JPEG

A standard image compression format developed by the Joint Photographic Experts Group.

LL(k) Grammar

A type of grammar whose productions can be predicted by using k look-ahead tokens.

Object-Z

An object-oriented extension of the Z formal specification language.

Parser

An application that is capable of determining whether a sequence of tokens belongs to a specific grammar.

PDF

Portable Document Format – A document format from Adobe© that can be displayed on many systems using the Acrobat Reader©.

WYSIWYG

"What You See Is What You Get" – A term used to indicate that an editor displays the content as it would appear on the printed page.

1 Introduction

Specifications are used to capture and communicate various aspects of a software product and can be used to guide analysis, development, and verification activities. They can range from less formal (informal) to more formal descriptions. Generally, specifications are refined from informal to formal during the planning and elaboration phases of a software project.

Informal specifications are generally written using natural languages and using graphics. Often, specification writers use terms or graphics that are not well defined or understood by all of the intended stakeholders which can lead to ambiguities or misunderstanding of the specification.

Formal specifications use a more precisely defined, mathematically based, language and are thus less likely to contain ambiguities because they have well defined syntactic and semantic rules. Ambiguities may still arise within a formal specification if there are syntactic or type errors. Syntax and type checking tools help reduce the possibility that a formal specification contains these errors. The rest of this report describes the development of such a tool for the Object-Z formal specification language.

The Object-Z language is an object-oriented extension of the Z specification language. Z was developed at the Oxford University in UK [10][11] . While Z is easy to use and is so popular both in academia and in industries, it is a procedural language. Since the object-oriented approach has become the *de facto* standard for software development, one would obviously look for an object-oriented formal specification language. Among the other object-oriented extensions of Z, the Object-Z language [7] developed at the University of Queensland, Australia became popular. This is because Object-Z was strongly supported by software industries in Australia and hence a lot of industrial case studies are available for references. Other object-oriented extensions of Z include ZEST and MOOZ [4].

One of the major problems in adopting formal methods in software development process is its steep learning curve. Part of this problem can be alleviated by providing adequate tool support. While many tools have been developed for Z, the language from

which Object-Z is derived, there are only fewer tools available for Object-Z. It is therefore evident that a WYSIWYG editor is needed that can also perform syntax and type checking. The desire is that a tool like this will make it easier for students and professionals to create and validate Object-Z specifications.

Currently, most of the specification languages are supported by LaTeX based tools in which an offline editor based on the LaTeX word processing system is used. This is because LaTeX supports an extensive set of mathematical symbols and macros for complex mathematical structures which are often part of any specification language. Such an approach uses built-in LaTeX macros to specify the specification constructs. A text editor is used to create a LaTeX document which is then compiled, using a LaTeX compiler, into a format suitable for display such as PDF. This method does not allow the user to immediately see what the rendered specification would look like. Also, in order to perform syntax and type checking, the user is required to save the specification and use other tools outside of the text editor.

While LaTeX is a popular document markup language, it is very likely that students learning Object-Z in a formal specification language course would not be proficient enough in LaTeX to develop correct Object-Z specifications . Learning the LaTeX language, the associated documentation preparation tools, and syntax and type checking tools would be an additional burden that distracts from the main focus of the course. The same distraction would be present for professionals who wish to adopt Object-Z to formalize critical areas of software development and testing. These concerns about tool support for Object-Z motivated the author to develop a WYSIWIG tool for the language. This tool is called TOZE, a The Object-Z Editor.

TOZE is a GUI-based tool that allows an Object-Z specification to be created and checked for syntax and type errors. The editor displays graphical elements of the language (such as the lines for the class paragraph) and provides text areas to enter textual and symbolic information. The text areas are positioned in appropriate places within a paragraph. Menu options allow the user to add, remove, and modify Object-Z paragraphs and icons can be selected to add symbols. In order to use TOZE, there is no need for the user to learn a markup language to create an Object-Z specification.

Syntax and type checking is performed within the tool without the need to save the specification and switch to another application. Errors are displayed using multiple cues to help easily identify an errors when they occur and to help the user locate them. Specifications can be exported as JPEG images and as LaTeX documents. Outputting the specification as LaTeX allows the user to render the document with more formatting controls and allows it to be included in other LaTeX documents. It was decided to implement the editor in Java to allow it to be run on a wide variety of platforms. A modified version of the Z fonts created by Lubos Mikusiak was used to display the symbols within Object-Z. More symbols were added to TOZE since Z fonts was only intended to support the Z specification language.

2 Existing Object-Z Tools

There are not many tools available for the creation and validation of Object-Z specifications. The ones that are known are either too old or relatively harder to use. Below is a list of tools available for Object-Z.

2.1 Wizard

Wizard is a syntax and type checking tool developed at and maintained by the Software Verification Research Centre at the University of Queensland in Australia [8]. Like many other tools for formal specification languages, Wizard takes as input the LaTeX formatted specification and outputs messages to standard output. Users of Wizard are expected to be familiar with LaTeX. Moreover, since LaTeX is an offline word processing language, the users will not be able to see the specification until it is compiled by a LaTeX processor. In addition, the users need to compile a specification with Wizard to see the syntax and type errors, and then need to compile the specification again with LaTeX processor in order to see the actual output. Another restriction is that Wizard works only on UNIX based platforms. Being developed by the same group who developed the Object-Z language, the tool captures most of the errors in Object-Z specifications. However, some of the error messages are not easily comprehensible. Wizard only supports Object-Z specifications unlike some other tools (listed below) that support both Z and Object-Z.

2.2 Moby/OZ

Moby/OZ is part of a suite of tools developed by the Correct Systems Design Group of the Department of Computing Sciences at the C.v.O University of Oldenburg [1]. It is advertised as a graphical editor that can be used to build Z and Object-Z specifications. At this time though, the product is not available.

2.3 CZT

The Community Z Tools (CZT) project is an open-source Java framework for building formal methods tools for Z and Z dialects [9]. Part of this project is the Object-Z Parser module. It provides a set of classes for parsing and printing Object-Z specifications. These classes have been used to create plug-ins for the jEdit and Eclipse editors [12] [13]. The editors allow text to be entered using the LaTeX markup or Unicode characters selected from a graphics panel. Graphical elements (boxes) are represented using Unicode characters to allow the user to work with the specification in a close approximation of how it would be rendered. Errors are displayed in ways typical to the specific editor.

2.4 ZML

ZML is a tool set based on XML/XSL that support Z, Object-Z, and TCOZ (Timed Communicating Object-Z) specifications. It was created by the School of Computing at the National University of Singapore [5]. It allows users to create Object-Z specifications in XML and view them using a Web browser. Like the Wizard tool expects the users to know LaTeX, the ZML tool users need to be familiar with XML. In addition to supporting Z and Object-Z, this tool also supports TCOZ, an extension of Object-Z that includes temporal constraints.

2.5 Z/EVES

Z/EVES is an interactive system for composing, checking, and analyzing Z specifications [14]. While it does not support Object-Z, it is an excellent example of a rich editor with a lot of features. Users are able to work with Z specifications in a WYSIWYG manner. In addition, users are not only able to perform syntax and type checking but can also use theorem proving capabilities. Instead of creating or editing a paragraph within a specification, the editor allows to create or edit a paragraph in a separate window and then merge it with the current specification at appropriate place. Symbols are added to paragraphs by using a palette of Z symbols. In addition to checking the entire specification, individual paragraphs can be checked separately. Errors associated with a specific paragraph are displayed in a separate popup window.

3 Software Development Methodology

An Agile software development methodology was used in developing the TOZE editor. The Agile methods emphasize face-to-face communication, especially with the customer representative, over written documentation [6]. The requirements for this project can be divided into two classes; those related to the Object-Z language and those related to the user experience. Those related to the language were well defined and consist of the grammar and type checking rules. Those related to the user experience were not as well defined and required input from the customer representatives. Because we were using a word processor as the model for the editor and a small team size, the developer felt that an agile methodology would be appropriate for this project.

The sponsors Dr. Kasi Periyasamy, and Dr. Kenny Hunt, acted as the customer representatives and provided feedback throughout the development of the tool. The developer had discussions with the sponsors before implementing each specific feature so that there was a common understanding between the customer representatives and the developer.

The basic vision for the tool was to create an Object-Z specification editor along the lines of a word processor. The entire set of functionalities was defined, developed and delivered in multiple releases. Feedback from earlier releases was incorporated into future releases. This is in contrast to a traditional waterfall development model where all the functionalities are defined upfront and delivered in one release.

The functionalities of the editor include the following (in no particular order):

1. Add, remove and edit simple specification paragraphs
2. Add, remove and edit remaining specification paragraphs
3. Save and load specifications
4. Check for and display syntax errors
5. Export specification to JPEG image and print
6. Check for and display type errors
7. Export specification as LaTeX document

4 Syntax Checking

Syntax checking is the process of analyzing a given input to determine if it conforms to a specific grammar. This is accomplished by transforming the input into a sequence of tokens and processing that sequence through a parser. A parser implements an algorithm that is able to recognize whether a sequence of tokens belongs to the class of languages defined by a specific grammar.

A parser for TOZE was created to recognize the language defined by the grammar specified in [7]. Since Object-Z is an extension of Z, the Z grammar as specified in [3] was implemented as needed to fully support Object-Z.

The parser implemented in TOZE is a recursive decent parser with backup. A recursive descent parser has a parsing procedure for each non-terminal production in the grammar it implements. A recursive descent parser with backup determines which productions to use by trying each possible right-hand side of a production without attempting to predict which one to use based on look ahead values. If a right-hand side of a production is tried and fails, the state of the parser is reset and the next alternative is tried. This type of parser does not require the grammar to be an LL(k) grammar as would a predictive parser. The developer chose this method in an attempt to have the parsing procedures mirror the grammar as much as possible and to eliminate the need to rewrite the Object-Z grammar to be LL(k).

Even though a grammar does not need to be LL(k) for a recursive descent parser with backup, it cannot have left recursion. Left recursion is a problem since it would cause a parsing procedure to call itself and result in infinite recursion. It occurs when the left-hand symbol of a production matches the first symbol of the right-hand side of the same production (direct left recursion) or if the first symbol causes the left-hand symbol to be matched without first matching to a non-terminal (indirect left recursion).

An example of direct left recursion appears in the following production:

A ::= A α

The parsing procedure for the above production could look something like:

function A()

{

```
    A();
    match('+');
  }
```

An example of indirect left recursion appears in the productions:

A ::= Bα

B ::= C

C ::= Aγ

The production A can be rewritten as

A ::= Cα

And then as

A ::= Aγα

which would cause infinite recursion as demonstrated above.

The Object-Z grammar as defined in [7] does not have indirect left recursion but it does suffer from direct left recursion. An example of this can be seen in the production

Expression4 ::= Expression4 . VariableName

Direct left recursion can be removed by creating a new non-terminal in the following way. Given the following production

A ::= Aα

A ::= β

replace "A" with:

A ::= βA'

Create a new production for "A'" as:

A' ::= αA'

A' ::=

Common prefixes is also a problem for recursive descent parsers with backup. Common prefixes occur when two productions with the same left-hand side have right-hand sides that have the same first symbol. This is a problem since the parser may match a shorter production than it otherwise could have.

For example, given the productions:

A ::= α

A ::= αβB

and the input αβ, the parser could match with the first production and then generate an error because there would be additional input. This was solved in the TOZE parser by ordering the productions so that the longest productions are tried before shorter productions.

During parsing, the longest parse not reaching the end of input is recorded. If no production resulted in the entire input being parsed, then a syntax error is generated with the location of the syntax error corresponding to the longest production matched. This is done to ensure that the longest sequence of tokens is recognized as valid input (i.e shortest syntax error). Only one syntax error will appear within an input at a time. The syntax checker does not attempt to recover from the syntax error and continue with any remaining input. Currently, this seems to be a limitation because the user will not be able to see all the errors in the specification. This problem will be taken care of in future releases of TOZE.

The Object-Z grammar contains multiple productions that would match a given input. An attempt was made to identify these ambiguities and modify the parser to manage them. One such ambiguity occurs with the following production

Expression4	::=	VariableName *[* ActualParameters*]*
	|	ClassName *[* ActualParameters*] [*RenameList*]*
	|	SchemaReference
VariableName	::=	Identifier
ClassName	::=	Word
SchemaReference	::=	SchemaReference1 *[* RenameList *]*
SchemaReference1	::=	SchemaName Decoration *[*ActualParameters *]*
SchemaName	::=	Word
Identifier	::=	Word Decoration
Decoration	::=	*[* Stroke … Stroke *]*

As can be seen in the above productions, a Word can be interpreted as a VariableName, SchemaName, or as a ClassName. When an expression is simply a Word, it is not possible to know whether the Word represents a variable name, schema name, or

class name. This determination needs to wait until the type of the Word can be determined during type checking. This part of the grammar is rewritten as :

Expression4 ::= Word Decoration *[* ActualParameters*] [* RenameList *]*

There are built-in functions within Object-Z which needed to be accommodated for in the grammar. These include **dom**, **ran**, **rev**, **head**, **last**, **tail**, **front**, **first**, **second**, and **#**. They were added to the Expression2 production as:

Expression2 ::= BuiltInFunctionName Expression4

where BuiltInFunctionName is the function name.

The Object-Z grammar did not specifically indicate where predefined data types should be included. Since the predefined data types can be used as an expression within a declaration, it made sense that some part of the grammar associated with expressions should be modified to include them. All of the predefined data types represent sets and so it made sense to add them to the SetExpression production.

The cardinality operator (#) can be used both as a prefix operator and as an infix operator. It is used as a prefix operator when returning the count in a set or sequence:

#S

It is used as an infix operator when determining how many items of a certain type are contained within a bag:

B#x

Because the editor allows expressions to span lines, this duality of the cardinality operator causes an issue with the parsing. Take for example, the following:

x < y

```
#z > 1
```

This may look like a conjunction of two predicates. The first ensures that "x" is less than "y" and the second ensures that the number of items in "z" is greater than one. Since the parser allows expressions to span lines, it is also possible to interpret the sequence of "y", "#', and "z" as retrieving the number of "z" items from the bag "y". This would leave ">" and "1" unparsed and cause a syntax error to be generated. To resolve this, the parser was written to interpret the # symbol as an infix operator only if the left-hand expression appears on the same line as the operator. Thus,

```
B
#x > 1
```

would be invalid if this was written to ensure that the number of "x" items in the bag "B" was greater than one, since "B" does not appear on the same line as the operator. The following would be valid:

```
B#x
> 1
```

since the left-hand expression appears on the same line as the operator even though the predicate it is used in spans multiple lines.

There are elements of the specification which, strictly speaking, do not need to be parsed. These are the graphical elements of the specification whose proper usage is enforced by the editor. These elements provide a framework around which the free-formatted text is structured. Only the text elements need to be parsed since the graphical elements of the language are known.

During parsing, an Abstract Syntax Tree (AST) is created. Some of the non-leaf nodes represent structural elements and are derived immediately from the framework elements. Others are created from parsing the text areas. The parsing of a text element results in an

AST which is then added to the node of the AST representing the structural element containing the text element.

5 Type Checking

Type checking is the process of ensuring that identifiers are declared and that operand types in expressions are compatible with corresponding operators. The input to type checking is the abstract syntax tree (AST) created by the parser. Type checking only occurs if the entire specification is successfully parsed (i.e. there are no syntax errors).

The abstract syntax tree is created during parsing. In the tree, the nodes represent constructs of the parsed input. The node at the root of the tree represents the entire specification and the descendent nodes represent paragraphs, operators, expressions, and other Object-Z constructs. There are many different types of nodes and each type of node encapsulates the knowledge of what types are valid for the specific construct it represents.

In the fully constructed abstract syntax tree, each node is an object of a specific type. Type checking is performed by asking a node to check that the types of its children are correct. If a node represents an expression then it asks its children to check their types and then return the type of the expression. This may involve using the types of its children if they are expressions. Otherwise, the node will just ask its children to check their types. To initiate type checking, the root node (which represents the entire specification) is asked to check its type. This causes the object to ask each of its children (global paragraphs) to check their types. This continues until all of the nodes have been checked.

Type checking is performed in two passes. The first pass identifies the user defined types and assigns types to variables and literals. The second pass ensures that variables and literals are being used in a manner consistent with the operations and types used in the operations. This allows for identifiers to be used before they are declared (forward declaration).

As described above, it is during the type checking process when the ambiguousness of an expression such as

Expression4 ::= VariableName *[* ActualParameters*]*

| ClassName *[* ActualParameters*] [*RenameList*]*

| SchemaReference

is resolved. Remember that the above production is rewritten as

Expression4 ::= Word Decoration *[* ActualParameters*] [* RenameList *]*

When this production is parsed, a node is inserted into the abstract syntax tree which indicates that the Word can be a variable name, schema reference, or class name. During type checking, the value of the Word symbol is looked up in the symbol table to determine which of these it is and that it is being used correctly.

The following expression:

x(1)

may be interpreted in two ways. First, it is interpreted as a function call with 1 as the parameter to the function "x". The second way is as an index to a sequence where the first element of sequence "x" is being accessed. The meaning is determined during type checking when the type of x is determined.

A single type error may cause other type errors within an expression. For example, given:

x > 1

if "x" is undefined that would cause one error. Not knowing the type of "x" may cause another error since "x" needs to be a numeric value in order to be used with the greater-than operator. It would be sufficient to simply indicate that "x" is undefined and not to indicate that both sides of the greater-than operator must be numbers. An effort was made to ignore type errors caused by other type errors in order to make it easier for the user to pinpoint the problem.

6 User Interface

This chapter describes the graphical user interface for TOZE. It explains the different options available for a user and shows how a specification is created, edited and checked for errors.

6.1 Main Window

The main window consists of the menu bar (1), specification window (2), and the messages window (3). See Figure 1.

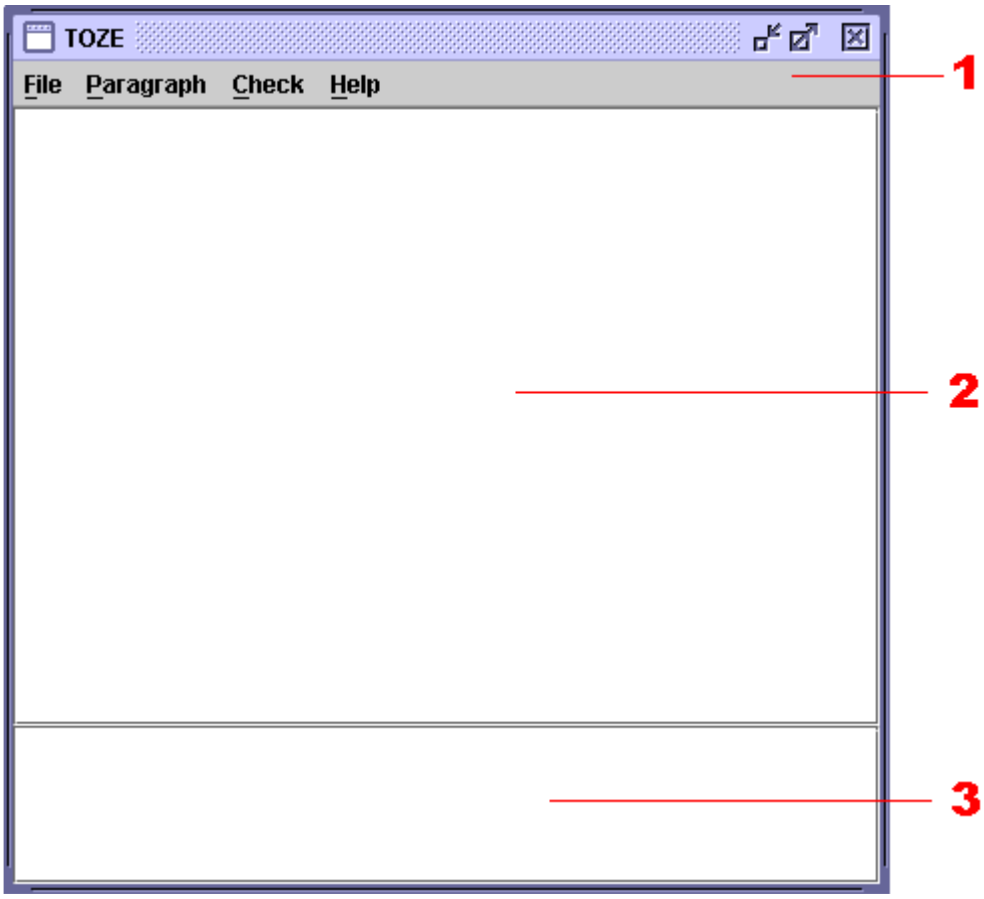

Figure 1. Main window

The specification window is where the specification will be displayed. Both vertical and horizontal scrollbars will appear as necessary if the specification window is not large enough to display the entire specification.

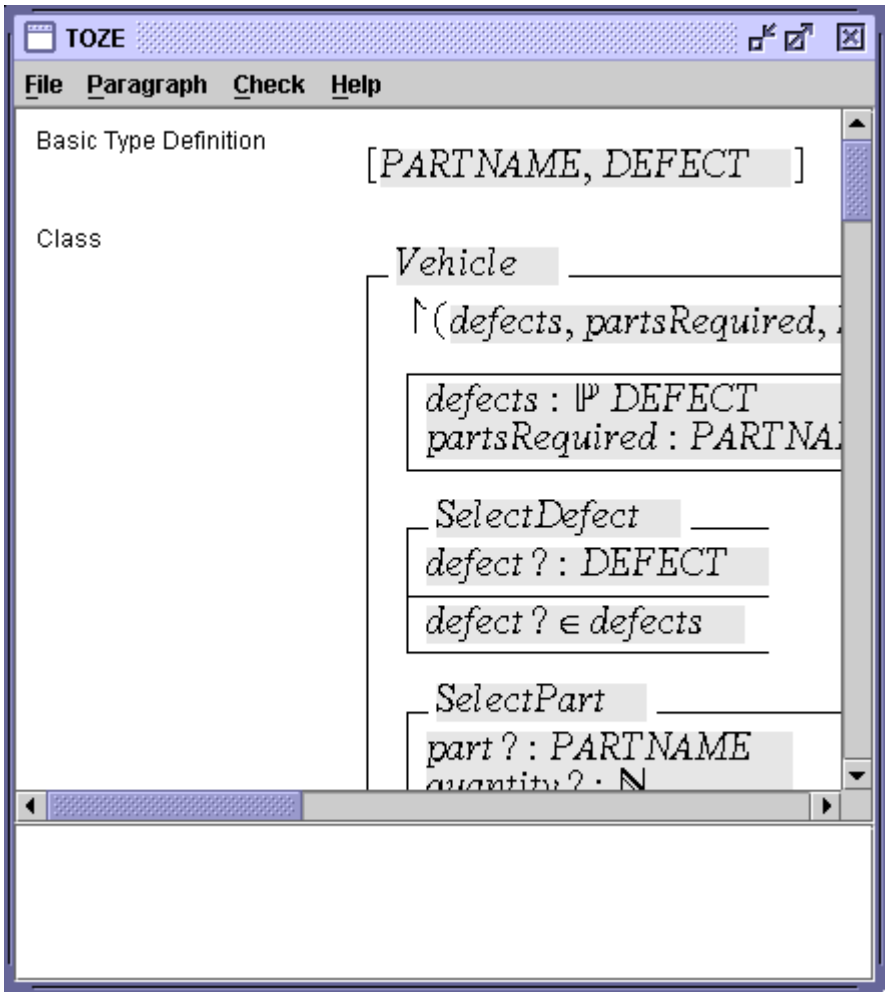

Figure 2. Scrollbars in the specification window

Paragraphs usually consist of structural elements and text areas. The structural elements are the lines used to surround and delimit areas of a paragraph. These structural elements vary for a given type of paragraph based on what is included in the paragraph. For example, if an axiomatic definition is added without any predicates, only a vertical line will appear to the left of the axiomatic definition. The horizontal line separating the definitions from the predicates will not be drawn. In Figure 3, the first axiomatic

definition contains a predicate whereas the second axiomatic definition does not contain a predicate.

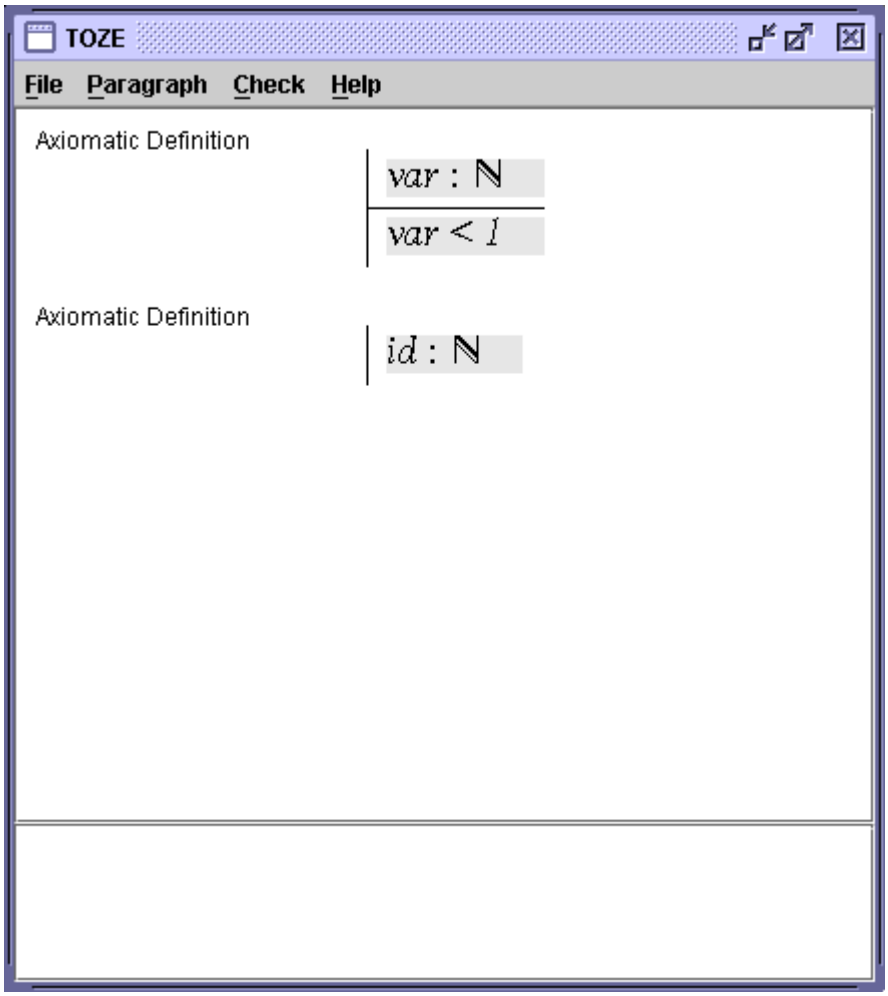

Figure 3. Axiomatic definitions with and without predicates

Text areas are used where the user is able to enter text and symbols. Text areas are identified by their light gray background.

On the left-side of each of the paragraphs, the type name of paragraph is displayed to aide in easy understanding.

6.2 Syntax Error Reporting

Figure 4 is an example of how TOZE reports a syntax error.

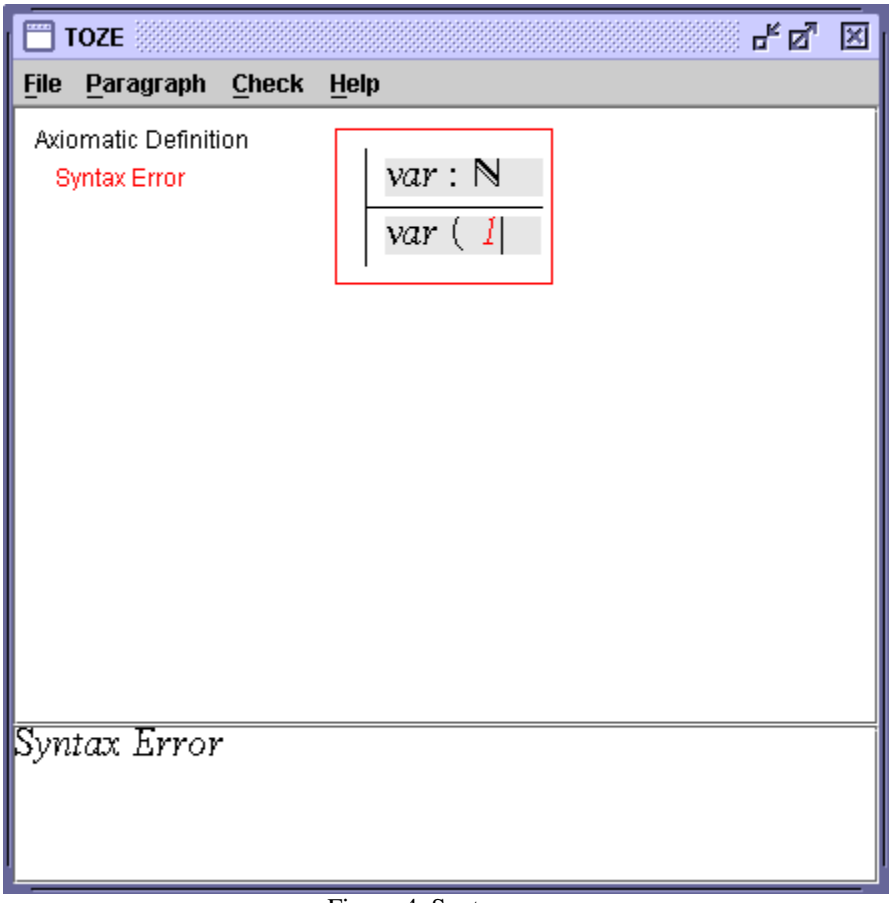

Figure 4. Syntax error

There are four types of indicators displayed when a syntax error is encountered.

1. The message window at the bottom of the screen will display the message “Syntax Error”.
2. The words “Syntax Error” will appear in red under the type name of the paragraph where the syntax error occurred.
3. A red box will be drawn around the paragraph where the syntax error occurred.
4. The offending text will be changed from black to red.

Individual text areas may be very long and the offending text may be out of view. The purpose of indicators 2 and 3 are so that it is easy to find which paragraphs contain the text area with the syntax error.

Each text area is parsed independently and it is possible that multiple text areas can report multiple syntax errors.

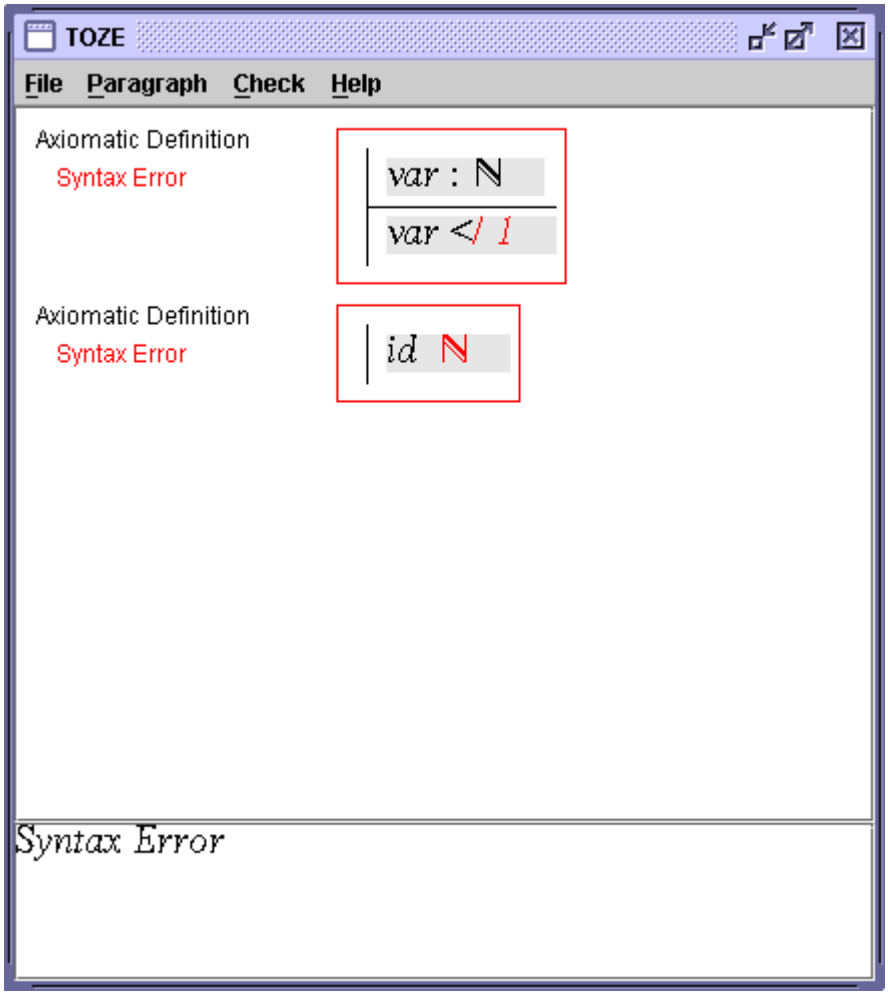

Figure 5. Multiple syntax errors

6.3 Type Error Reporting

Figure 6 is an example of how TOZE reports type errors.

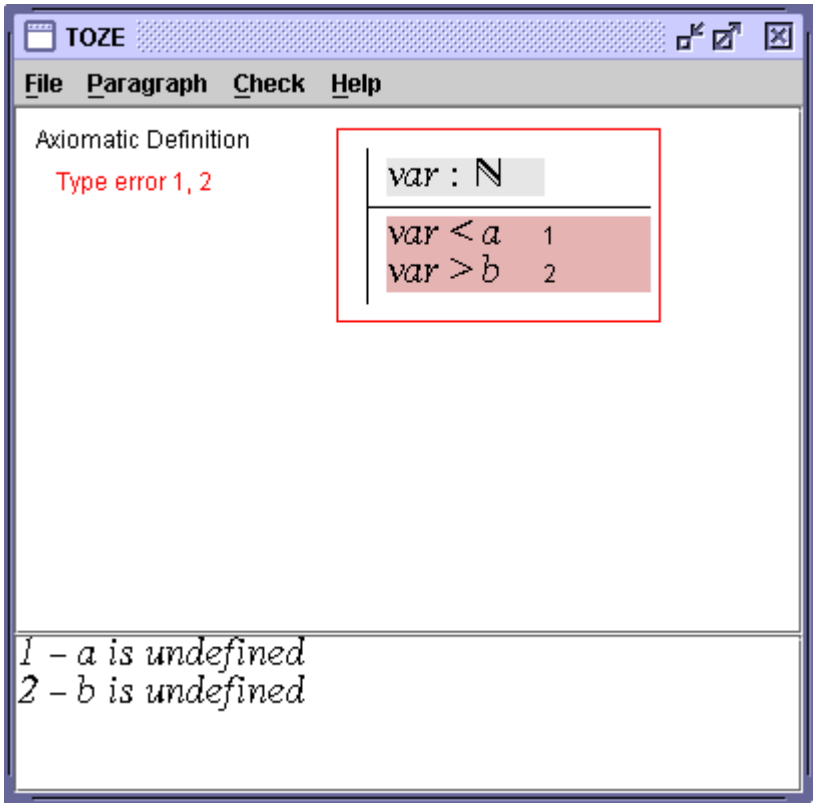

Figure 6. Type error

Five types of indicators are displayed when a type error occurs.

1. A message is displayed in the message window for each type error found. Each error is assigned a unique number which is displayed with the message.
2. The phrase "Type error" will appear in red under the type name of the paragraph where the type error occurred. In addition, the numbers corresponding to the type errors found in the paragraph are displayed next to the phrase "Type error".
3. A red box will be drawn around the paragraph where the type error occurred.
4. The background of the text area containing the type error will be turned to a light shade of red.
5. The numbers corresponding to the type errors found within a text area are displayed within the text area on the right-most side and on the line where the type error occurs.

6.4 Symbol Window

Almost every formal specification language uses a lot of mathematical symbols. It is a hard part of the design of a tool for a language to provide a mechanism for the user to type in these special symbols. LaTeX has a rich set of symbols and macros and hence is a preferred medium of developing formal specification tools. However, in a WYSIWIG editor such as TOZE, the problem is cumulative. The users are expected not to memorize any command or procedure to type in the specifications. TOZE provides an extensive set of symbols used in Object-Z. Unlike LaTeX based tools which require special fonts to be loaded in the computer system, TOZE is distributed with a font file that is loaded into the editor at runtime. Both text and symbols can be entered into the text areas. One way users may enter symbolic information into a text area is by using the symbol window. The symbols are listed along with their associated keyword. Clicking on the symbol will cause that symbol to be included in the text area at the active cursor location. Alternatively, a user can type in the keyword for the symbol (indicated on the left column of the table in Figure 7). As soon as the user types in the last character of the keyword, the corresponding symbol appears in the text window. This option would be easier for users once they become familiar with the keywords.

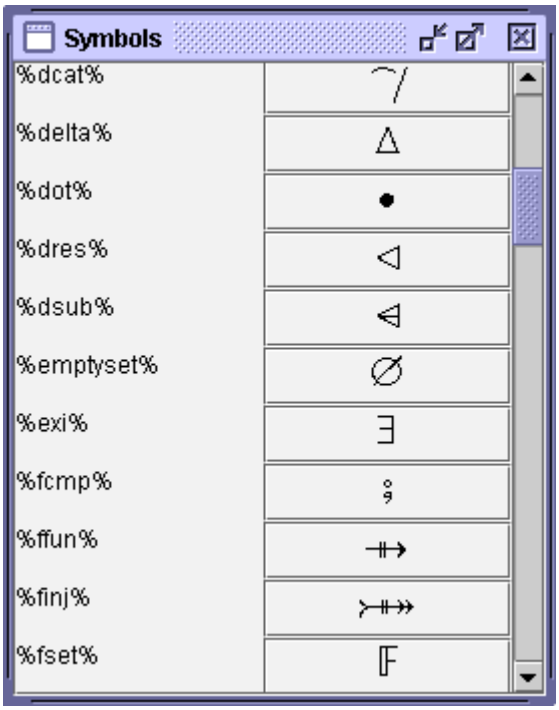

Figure 7. Symbol window

7 User Operations

This section assumes that the reader is familiar with Object-Z specification. Interested readers can refer to [7] for a detailed syntax of Object-Z language.

7.1 Working with Paragraphs

An Object-Z specification consists of several paragraphs of, possibly, different types. Paragraphs are added by using the Paragraph menu. Figure 8 illustrates how a user can select a paragraph type and include it as part of the specification.

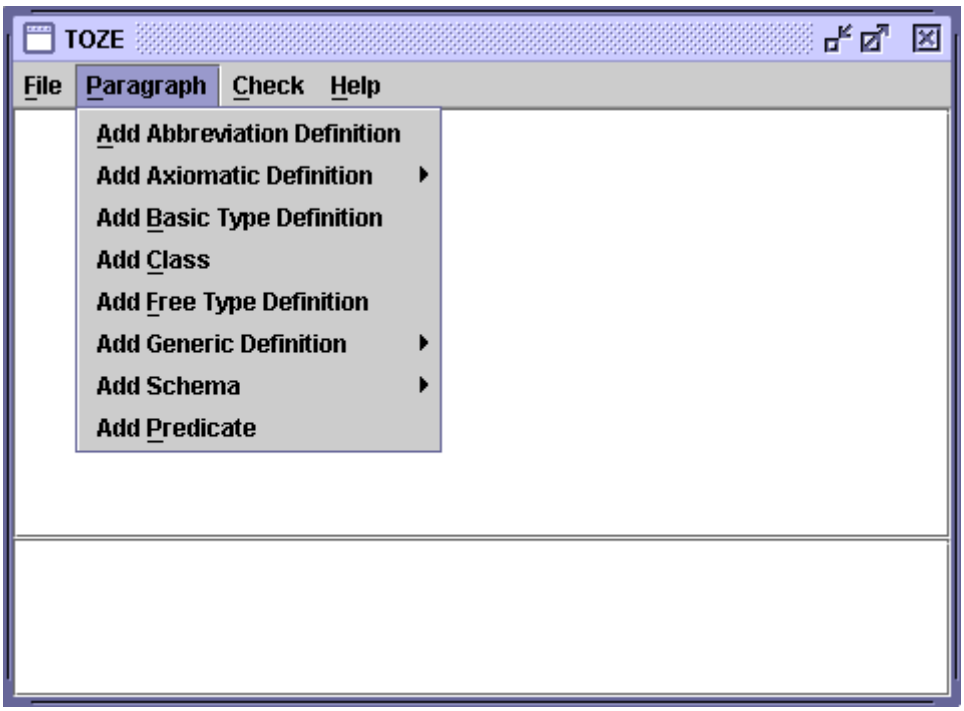

Figure 8. Paragraph menu

Some paragraphs can take different forms and can be added directly in the modified form from the Paragraph menu. The axiomatic definition, generic definition, and schema paragraphs can be added with or without a text area for predicates. Also, a schema can be added as a schema expression.

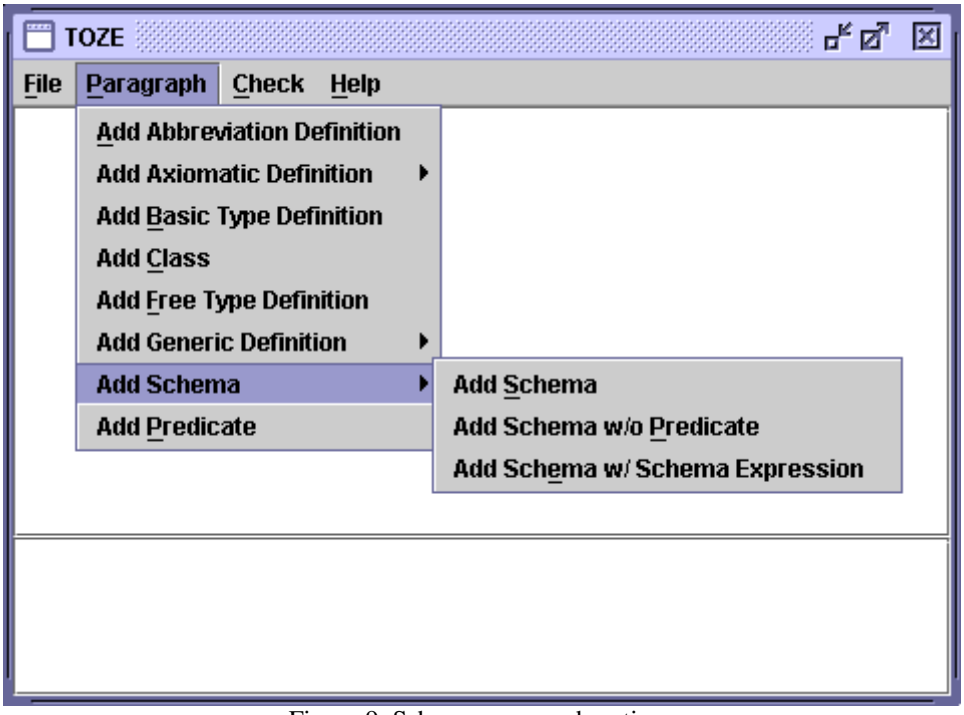

Figure 9. Schema paragraph options

When a paragraph is added to a specification, it is added to the end of the current specification. There is currently no functionality to insert paragraphs before existing ones.

Once added, paragraphs can be modified at any time. Right clicking on a paragraph will display a popup menu with appropriate options to modify that paragraph. The items available on the popup menu sometimes depend on where within the paragraph the mouse click is. Take for example, an axiomatic definition with a predicate. Right clicking anywhere on the paragraph outside of the predicate text area results in the following popup menu:

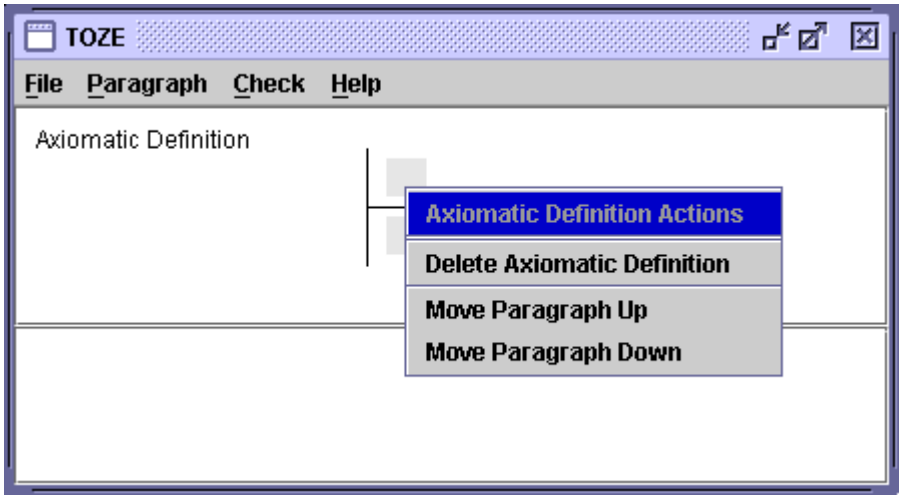

Figure 10. First type of popup menu

The options available are to delete or move the paragraph. Right clicking within the predicate text area results in the following popup menu:

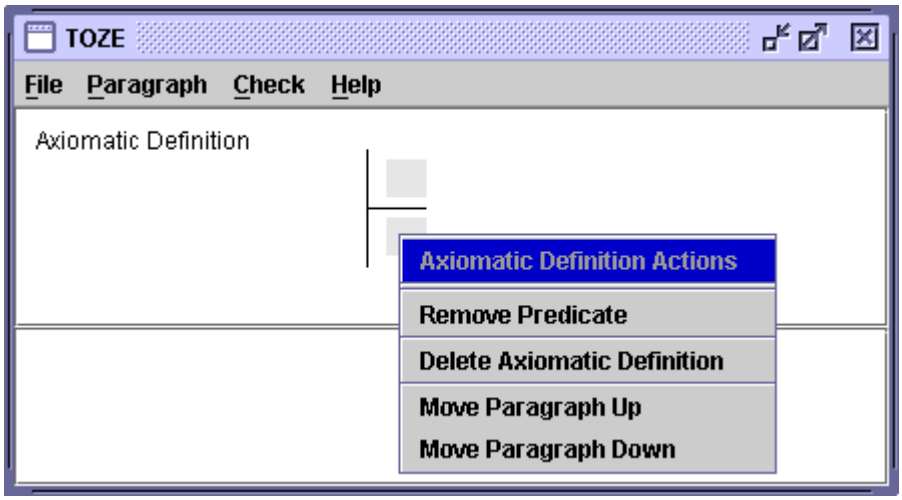

Figure 11. Second type of popup menu

Now there exists an option to remove the predicate text area which would result in an axiomatic definition with only a text area for definitions. If the predicate text area is removed the right clicking anywhere within the paragraph would result in the following popup menu:

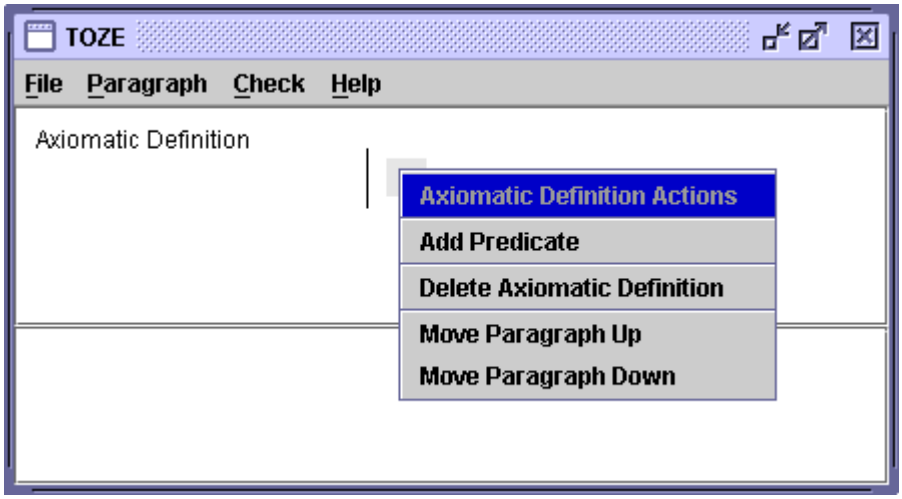

Figure 12. Third type of popup menu

Now there is an option to add the predicate text area. This option does not appear in the popup menu if the predicate text area exists since only one predicate text area is needed. Along with the axiomatic definition paragraph, the generic definition and schema definition paragraphs work this way.

The class paragraph allows for many more options than the other paragraphs since there are many other elements that can appear within the class paragraph. See Figure 13 for options within a class paragraph.

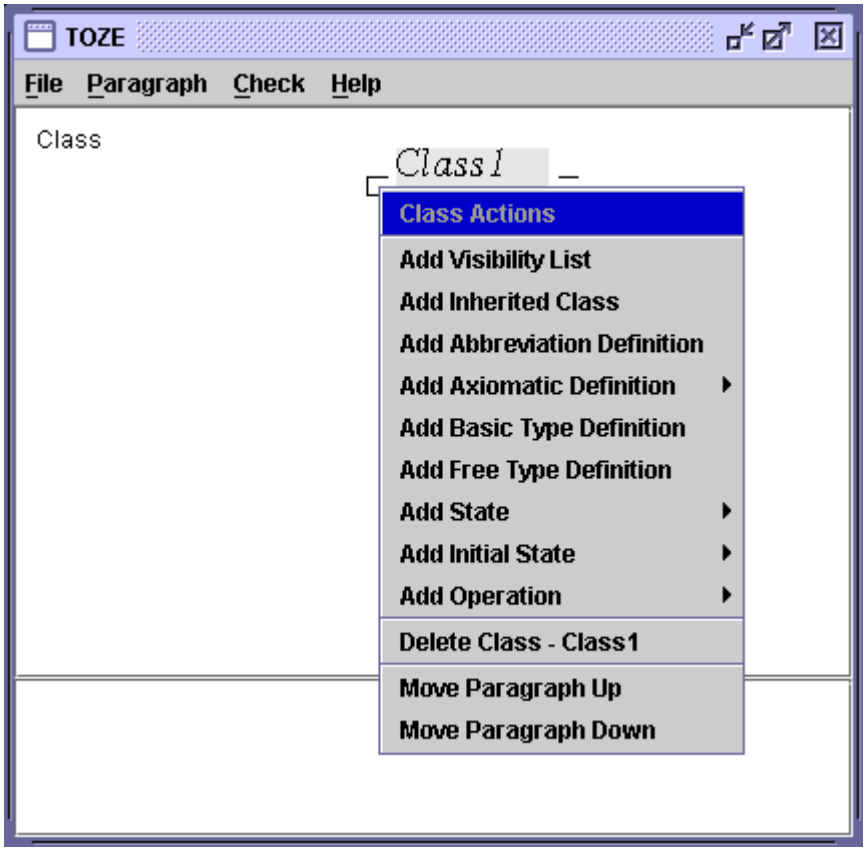

Figure 13. Class popup menu

Options within the popup menu for the class paragraph will not appear if it is not valid to add it. For example, if the class paragraph contains a visibility list, the popup menu would not contain an "Add Visibility List" option since it is not valid to have multiple visibility lists.

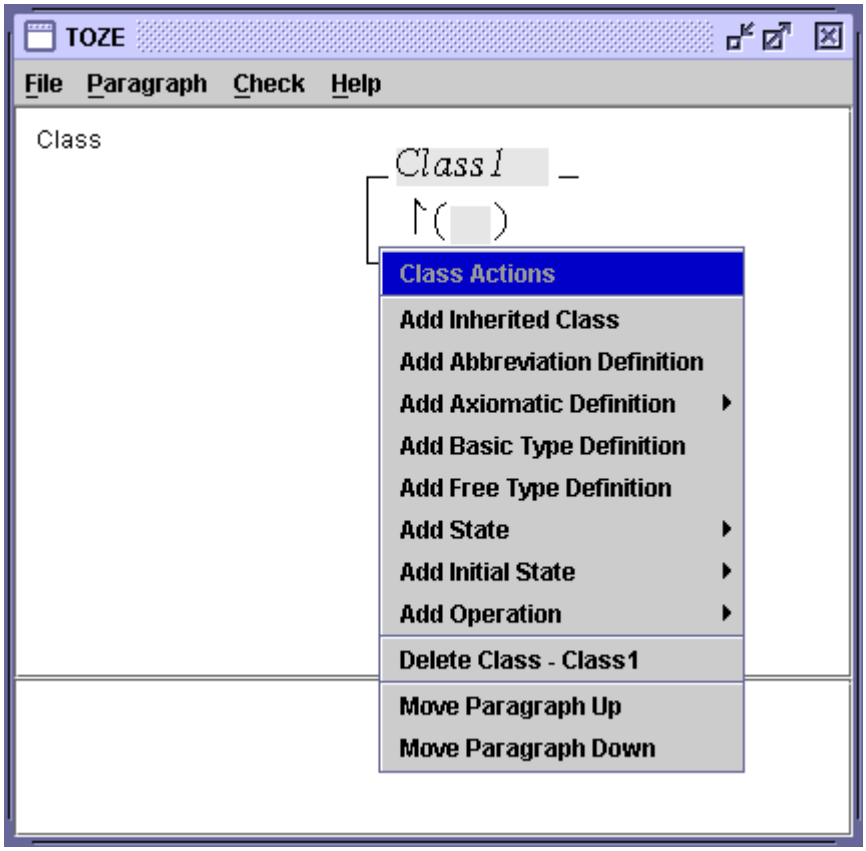

Figure 14. Class popup menu with visibility list

When adding class paragraphs, the editor ensures that the class paragraphs are displayed in the proper order regardless of the order they were added. For example, visibility list will be displayed before any local definitions even if a local definition was added before the visibility list. Multiple local definitions will appear together in the order in which they were added.

Right clicking within a class paragraph will display a popup menu with items specific to that class paragraph. For example, it is possible to add or remove the predicates from an axiomatic definition so the appropriate “Add” or “Remove” predicate menu item will appear in the popup menu along with an item to delete the axiomatic definition. The inherited classes paragraph cannot be modified in that way and its popup menu will only include an item to delete it.

Each paragraph can be moved up or down by selecting the corresponding item in the popup menu. Visual cues will remain visible if they were present in the paragraph before it was moved. Also, the numbering of the type errors will remain the same.

7.2 Editing Text

Within each paragraph exists text areas where text and symbols may be entered. Letters and numbers may be entered using the keyboard and symbols may be entered in two ways. The first option is to use the Symbol Window by clicking on the desired symbol. The second option is to enter the symbol's associated keyword delimited by percent signs (%). Once the final percent sign is entered, the keyword and associated percent signs are converted to the proper symbol. The symbol keywords were chosen to coincide with the related LaTeX macro for that symbol so that LaTeX users can easily remember the keywords. .

The active text area is the text area where characters and symbols are inserted when typed or selected from the symbol window respectively. The active text area is identified by the presence of the cursor. The active text area can be changed by clicking within the desired text area or by hitting the tab key to make the next text area active.

Text areas can be single or multiline. Text areas used for definitions and predicates are multiline text areas; all others are single line text areas. Multiple definitions can appear within a definition text area but may only start on a new line or begin with a semicolon if beginning on a line after another definition. A definition may also span across multiple lines as illustrated in Figure 15.

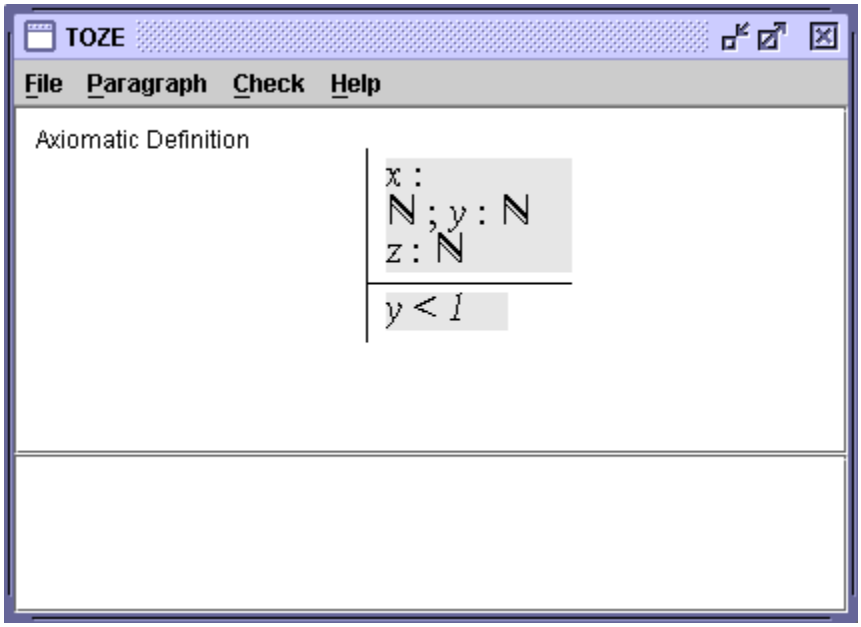

Figure 15. Definitions spanning lines, on same line, and separate line

In figure 15 , the definition text area contains multiple definitions and one that spans multiple lines. The definition for “x” spans multiple lines. The definition of “y” begins on the same line of the definition for “x” and thus begins with a semicolon (;). The definition of “z” appears on its own line.

The same holds true for predicates. Multiple predicates can appear within a predicate text area but may only start on a new line or begin with a semicolon if beginning on a line after another predicate. A predicate may also span across multiple lines. Figure 16 contains an example for this scenario.

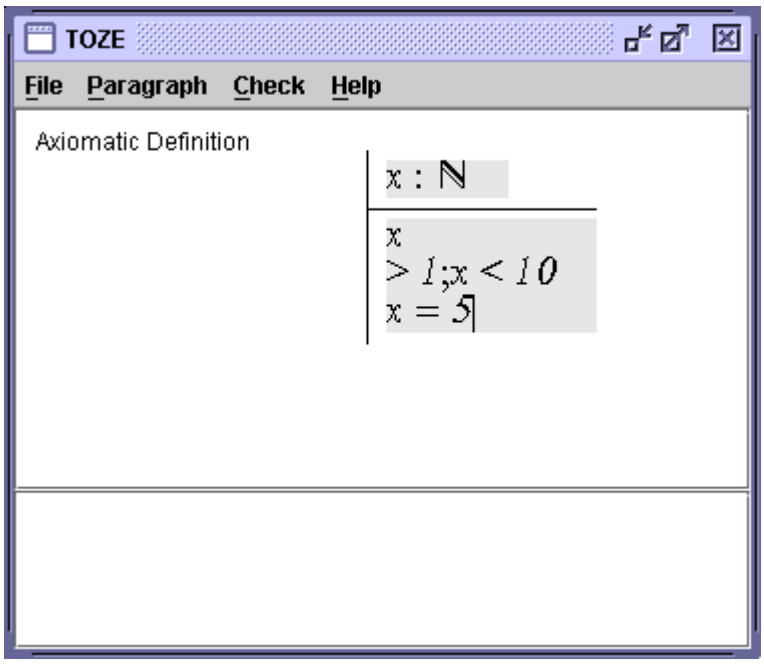

Figure 16. Predicates spanning lines, on same line, and separate line

If text is modified within a text area that either has a syntax or type error, the visual clues within the text area are removed. For instance, if the text area has a syntax error, the red text which generated the syntax error will turn to black. If the text area has a type error, the background will return to gray and the numbers within the text area will be removed. The reason for doing this is because it is possible that once the text is changed, any existing errors will not be valid. Rechecking the specification will redisplay the visual cues if those errors are still in the specification. The visual cues outside of the text area (e.g. the red box around the paragraph) will remain.

7.3 Checking the Specification

Syntax and type checking is initiated by selecting the “Check” item under the “Check” menu or by typing Ctrl-c.

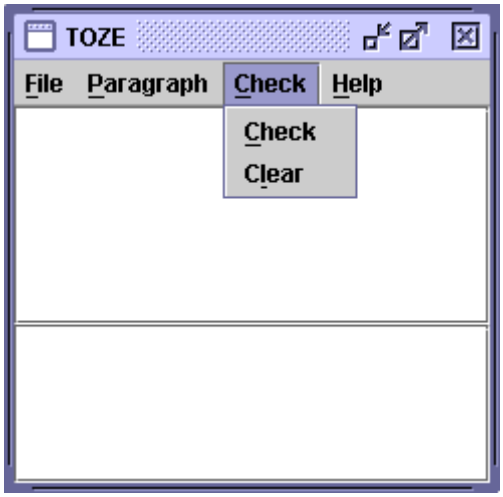

Figure 17. Check menu

Errors are identified by phrases in the message window and by visual cues within the specification. If there are syntax errors, the only message in the message window will be "Syntax Error". If there are type errors then the type errors will be displayed within the message window and visual cues will be displayed within the paragraph containing the type error. The text area containing the type error can be navigated to by double clicking on the number associated with the type error message within the message window. This will move the cursor to the location of the type error. If there are no errors, the phrase "No errors" will appear in the messages window. The message window will remain blank if the specification has not been checked.

The "Clear" menu item under the "Check" menu will clear all error messages from the message window and remove all visual cues from the specification window.

7.4 File Operations

The editor has file operations such as "new", "open", "save", "save as", and "exit" which behave as one would expect in a typical word processor application.

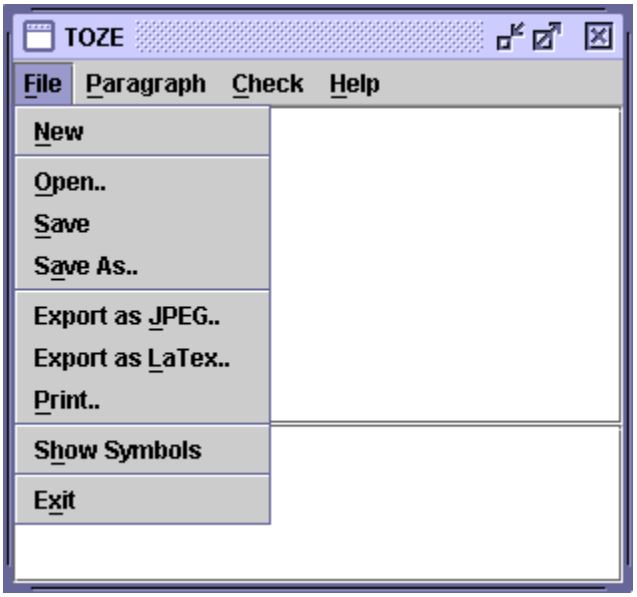

Figure 18. File operations

When the editor is first started, it contains an empty specification to which a user can add more paragraphs. There is no need to explicitly create a new specification. In this state, the specification does not have a filename associated with it and the user will be prompted for one upon saving the specification. Once a filename is associated with the specification, it may be saved without the user being prompted for a filename. When an existing specification is opened, the filename is implicitly associated with the specification. Whenever a filename is associated with a specification, the filename will appear in the title bar of the editor.

The editor keeps track of when changes are made to the specification. If changes were made and the user selects the “New”, “Open”, or “Exit” menu items, the user will be asked if they would like to save the changes before continuing with the operation. The user also has the option of canceling the operation.

The specification may be exported as a JPEG image or a LaTeX document. If a file type is not included when exporting the specification, a file type of “.jpg” is used when exporting as a JPEG image and “.tex” when exporting as a LaTeX document.

When a specification is exported as a JPEG image, any visual cues associated with errors will also be saved. No error information is included when the specification is exported as a LaTeX document.

Printing is accomplished by sending an image of the specification window to the printer. The image may be printed on multiple vertical pages but not horizontally. The image will be scaled as necessary to make it fit horizontally within the width of one page. It will be scaled vertically to maintain the aspect ratio and may span multiple pages.

8 Future Work

The tool provides all of the proper functionality to create and check Object-Z specifications. There are some areas of the tool that could be improved to make the user experience better. Some of these enhancements are described in this section.

Navigating within the specification could be improved. Page Up and Page Down keys are not implemented and could be used to allow the user to move through the document a screen at a time. Currently, only the Tab key can be used to move forward or backwards a single text area at a time. It may also be useful to allow the user to move from one text area to another using the arrow keys instead of limiting them to just movement within a text area.

Paragraphs can be moved up and down by selecting the appropriate menu item from a popup menu. Being able to click on a paragraph and drag it to a new location would mirror the functionality in word processors of being able to select text and drag it to a new location.

Data within the text areas may be selected, copied, and pasted between text areas and other applications. This functionality is provided by default with the Swing Java classes. Extending this concept to paragraphs would allow entire paragraphs to be copied from one specification to another. A handy feature would be the ability to copy a paragraph from the specification and have it pasted into a LaTeX document as LaTeX commands.

Using LaTeX is a very common way of creating Object-Z specifications and is used often in academic settings. Being able to import LaTeX would allow users to import existing specifications to make changes versus having to reenter the specification in order to use the tool.

As noted above, only one syntax error per text area will be displayed if a parsing error is encountered. Various methods have been devised to continue past a parsing error in order to complete parsing of the input. A suitable method could be chosen and implemented in the parser to allow the user to see multiple syntax errors for a single text area and not wait until the first syntax error is fixed to see if more exist.

Even though a specification may not contain syntax or type errors it may not be correct. A theorem prover could be added to the editor to enable the user to determine the correctness of the specification. An excellent example of theorem proving functionality can be seen with [15].

Creating a formal specification is only one part of an entire software development life cycle. For example, once the specification is completed, it may be referenced during class diagramming activities or test case development. Adding functionality to the editor to automate class diagramming or test case development would help users leverage the work that was applied to the specification and reduce the chance of errors being introduced. This is similar to how some class diagramming tools are capable of generating source code.

9 Conclusion

This paper describes a WYSIWYG editor for the editing and validation of Object-Z specifications. Graphical elements of the Object-Z specification language communicate important information and being able to work with a specification has a big advantage over working with a specification in a markup language where the graphical elements are not as obvious. The way the user interacts with the tool will be familiar to anyone who has used a document-oriented application before, such as a word processor, and should allow the student or professional to very quickly begin taking advantage of the features provided in the editor. This familiarity will allow the user to focus on working with the specification and not be distracted with the need to learn LaTeX commands and tools.

The error checking capability are integrated within the editor and the resultant messages and visual cues are adequate in helping the user resolve syntactical and type errors present in the specification. This instant feedback will help keep the user focused on working with the specification and not burden them with the distraction of switching tools to accomplish what they need. This capability seems extremely valuable for the student who is beginning to learn Object-Z and for which these distraction would impede their learning.

10 Bibliography

1 Carl von Ossietzky Universität, Department für Informatik, Correct System Design Group, "Moby Homepage", [Online] Available: http://csd.informatik.uni-oldenburg.de/~moby/ [Accessed: Nov. 24, 2008].

2 Charles N Fischer, Richard J LeBlanc Jr, "Crafting a Compiler", The Benjamin/Cummings Publishing company, Inc., 1988.

3 Roger Duke and Gordon Rose, "Formal Object-Oriented Specification using Object-Z", Macmillan Press Ltd., 2000.

4 ISO/IEC 13568:2002(E), Information technology – Z formal specification notation – Syntax, type system and semantics.

5 Kevin Lano and Howard Haughton, "Object-Oriented Specification Case Studies", Prentice Hall, 1994.

6 National University of Singapore, "Z Family on the Web with their UML Photos", , [Online] Available: http://nt-appn.comp.nus.edu.sg/fm/zml/ [Accessed: Nov. 24, 2008].

7 Peter Schuh, "Integrating Agile Development in the Real World", Cengage Charles River Media, 2004.

8 Graeme Smith, "The Object-Z Specification Language", Kluwer Academic Publishers, 2000.

9 Graeme Smith, "Object-Z Tool Support", [Online] Available: http://www.itee.uq.edu.au/~smith/tools.html [Accessed: Nov. 24, 2008].

10 Sourceforge, "Community Z Tools", [Online] Available: http://czt.sourceforge.net/ [Accessed: Nov. 24, 2008].

11 Mike Spivey, "The Z Notation: a reference manual", Prentice Hall International (UK) Ltd., 1992.

12 Jim Woodcock and Jim Davies, "Using Z: Specification, Refinement, and Proof", Prentice Hall, 1996.

13 Mark Utting, "CZT Eclipse Plugin", [Online] Available: http://www.cs.waikato.ac.nz/~marku/czt/eclipse.html [Accessed: Nov 29, 2008].

14 Sourceforge, “CZT jEdit plugins”, [Online] Available: http://czt.sourceforge.net/jedit/index.html [Accessed: Nov. 29, 2008]

15 Mark Saaltink, “The Z/EVES 2.0 User’s Guide”, ORA Canada, 1999.

11 Appendix A. Modified Object-Z grammar

Below is the grammar as implemented by the parser. It is written using an extended BNF as used in [7].

---

nIdentifier ::= Identifier … Identifier

nBranch ::= Branch | … | Branch

ClassHeader ::= ClassName *[* FormalParameters *]*

nInheritedClass ::= InheritedClass … InheritedClass

The order of Abbreviation is important since a VariableName can be an Identifier. If VariableName was parsed first, then input matching Identifier InfixGenericName Identifier would match VariableName without the FormalParameters since the FormalParameters are optional.

Abbreviation ::= Identifier InfixGenericName Identifier
| VariableName *[* FormalParameters *]*
| PrefixGenericName Identifier

SchemaHeader ::= SchemaName *[* FormalParameters *]*

PredicateList ::= Predicate *[[* ; *] [eol]* Predicate *]*

OperationExpression ::= ∧ Declaration *[* | Predicate *]* • OperationExpression
| [] Declaration *[* | Predicate *]* • OperationExpression
| ⨾ Declaration *[* | Predicate *]* • OperationExpression
| rrOperationExpression1

rrOperationExpression1 ::= OperationExpression1 restrrOperationExpression

restrrOperationExpression ::= ∧ OperationExpression1
| ⨾ OperationExpression1
| [] OperationExpression1
| • OperationExpression1
| || OperationExpression1
| || ! OperationExpression1

The order of OperationExpression1 is important for the last two alternates since an Identifier can be part of an expression.

```
OperationExpression1   ::= [ DeltaList Declaration [ | Predicate ] ]
                         | [ Declaration [ | Predicate ] ]
                         | [ Predicate ]
                         | ( OperationExpression )
                         | Expression.Identifier [ RenameList ]
                         | Identifier [ RenameList ]

Expression             ::= if Predicate then Expression else Expression
                         | rrExpression1

rrExpression1          ::= Expression1 restrrExpression1

restrrExpression1      ::= InfixGenericName rrExpression1
                         |

Expression1            ::= rrExpression2 optExpression1

optExpression1         ::= × rrExpression2 optExpression1
                         |

rrExpression2          ::= Expression2 restrrExpression2

restrrExpression2      ::= InfixFunctionNameX rrExpression2
                         |

Expression2            ::= ℙ Expression4
                         | 𝔽 Expression4
                         | PrefixGenericNameX Expression4
                         | - Decorations Expression4
                         | BuiltInFunctionName Expression4
                         | rrExpression4 ⦇ Expression0 ⦈ Decorations
                         | rrExpression3

rrExpression3          ::= rrExpression4 restrrExpression3

restrrExpression3      ::= Expression3
                         |

Expression3            ::= rrExpression4

rrExpression4          ::= Expression4 restrrExpression4
```

restrrExpression4 ::= ∪ Expression4 optrestrrExpression4
| ©
| . VariableName
| (Expression3)
| PostfixFunctionName
|

optrestrrExpression4 ::= restrrExpression4
|

Expression4 ::= WORD *[* Decorations *]* *[* ActualParameters *]* *[*RenameList *]*
| SetExpression
| (Expression , Expression *[* , Expression *]**)
| **self**
| NUMBER
| ↓ Expression4
| (Expression0)
| ⟨ Expression , … , Expression ⟩
| ⟦ Expression , … , Expression ⟧
| ∪ Expression

Expression0 ::= λ SchemaText • Expression
| μ SchemaText • Expression
| LET LetDefinition ; … ; LetDefinition • Expression
| Expression

SetExpression ::= { Expression , … , Expression }
| { SchemaText *[* • Expression *]* }
| ℕ
| $\mathbb{N}_1$
| ℤ
| B
| ℝ
| ∅

BuiltInFunctions ::= **dom**
| **ran**
| **rev**
| **head**
| **last**
| **tail**
| **front**

| **first**
| **second**
| #

Predicate ::= ∀ SchemaText • Predicate
| ∃ SchemaText • Predicate
| $\exists_1$ SchemaText • Predicate
| **let** LetDefinition ; … ; LetDefinition • Predicate
| rrPredicate1

rrPredicate1 ::= Predicate1 restrrPredicate1

restrrPredicate1 ::= ∧ rrPredicate1
| ∨OR rrPredicate1
| ⇒ rrPredicate1
| ⇔ rrPredicate1

Predicate1 ::= Expression Relation … Relation Expression
| Expression **.INIT**
| Predicate1sub1
| PrefixRelationName Expression
| **pre** SchemaReference
| **true**
| **false**
| ¬ Predicate
| (Predicate)
| SchemaReference

SchemaExpression ::= ∀ SchemaText • SchemaExpression
| ∃ SchemaText • SchemaExpression
| $\exists_1$ SchemaText • SchemaExpression
| rrSchemaExpression1

rrSchemaExpression1 ::= SchemaExpression1 restrrSchemaExpression1

restrrSchemaExpression1 ::= ∧ rrSchemaExpression1
| ∨ rrSchemaExpression1
| ⇒ rrSchemaExpression1
| ⇔ rrSchemaExpression1
| ↾ rrSchemaExpression1
| \ (DeclarationNameList)
| ⨾ rrSchemaExpression1
| >> rrSchemaExpression1

| |

SchemaExpression1 ::= [SchemaText]
| SchemaReference
| ¬ rrSchemaExpression1
| **pre** rrSchemaExpression1
| (SchemaExpression)

VariableName ::= Identifier Decorations
| (OperatorName)

Identifier ::= WORD Decorations

OperatorName ::= _ InfixFunctionName _
| InfixGenericName _
| InfixRelationName _
| PrefixGenericName _
| _ (| _ |) Decorations
| _ Decorations

DeltaList ::= Δ (DeclarationNameList)

ActualParameters ::= [Expression , … , Expression]

RenameList ::= [RenameItem , … , RenameItem]

FormalParameters ::= [Identifier , … , Identifier]

RenameItem ::= DeclarationName / DeclarationName

Declaration ::= BasicDeclaration ; … ; BasicDeclaration

BasicDeclaration ::= DeclarationNameList : Expression
| SchemaReference

DeclarationNameList ::= DeclarationName , … , DeclarationName

DeclarationName ::= Identifier
| OperatorName

SchemaText ::= Declaration *[* | Predicate *]*

SchemaReference ::= SchemaName *[* Decorations *] [* ActualParameters *] [* RenameList *]*

SchemaName ::= WORD

LetDefinition ::= VariableName == Expression

Relation ::= =
| ∈
| InfixRelationName

Decorations ::= ‘
| ?
| !
|

The order of the Branch production is important because Identifier is a subset of VariableName

Branch ::= VariableName《 Expression 》
| Identifier

ClassName ::= WORD

InheritedClass ::= ClassName *[* ActualParameters *]* *[* RenameList *]*

InfixRelationName ::= see [7]
PostfixFunctionName ::= see [7]
InfixGenericNameX ::= InfixGenericName
InfixGenericName ::= see [7]
InfixFunctionNameX ::= InfixFunctionName
InfixFunctionName ::= see [7]
PrefixGenericNameX ::= PrefixGenericName
PrefixGenericName ::= see [7]
PreficRelationName ::= **disjoint**

12 Appendix B. Type Error Messages

This is a list of type error messages produced by the editor. In the messages below, ***x*** represents an identifier and ***o*** represents an operator.

x is not defined

x must be a class

x is not a valid member and cannot be included in the visibility list

x is already defined

Expressions used in declarations must result in a set

The schema ***x*** does not exist

x is not a schema name

x is not defined for this operation

The operation ***x*** does not exist

x is not an operation name

Predicates must evaluate to a boolean value

Right side of ***o*** must be a set

Element is not the same type as the set for ***o***

Both sides of ***o*** must be sets of the same type

Both sides of ***o*** must be the same type

Both sides of ***o*** must be numbers

Right side of ***o*** must be a sequence

Left side of ***o*** must be a sequence

Both sequences of ***o*** must be of the same type

Type mismatch

The expression used for the then clause is a different type than the one used for the else clause

Both expressions of a cross-product must evaluate to a set

Both sides of an infix generic operator must be sets

The expressions for the ***o*** operator must be numbers

Expressions for the ***o*** operator must be sets

Sets must be the same type for the operator ***o***

Expressions for the ***o*** operator must be sequences

Expressions for ***o*** must be numbers

Expressions for ***o*** must be relations

Relations used in ***o*** must be of the same type

Left-hand argument to ***o*** must be a set

Right-hand argument to ***o*** must be a relation

Type of set must be the same as the domain type of the range

Right-hand argument to ***o*** must be a set

Left-hand argument to ***o*** must be a relation

Type of set must be the same as the range type of the relation

Left-hand expression for a projection must be a set

Right-hand expression for a projection must be a sequence

Power set must be applied to a set

Left-hand expression to the image operator must be a relation

Inside expression of the image operator must be a set

The type of the inside set expression must the the same type as the domain type of the relation for the image operator

Expression used in distributed union must be a set of sets

All sets used in a distributed union must be of the same type

Expression used in distributed union must be a set of sets

Expression used in distributed intersection must be a set of sets

All sets used in a distributed intersection must be of the same type

Undefined class variable

Member access must be from a class type

The member ***x*** is not visible

The member ***x*** is undefined

The argument to '~' must be a relation

The argument to ***o*** must be a relation

All expressions of a sequence must be of the same type

The argument to the function 'dom' must be a set of tuples

The argument to the function 'dom' must be a tuple of size 2

The argument to the function 'ran' must be a set of tuples

The argument to the function 'ran' must be a tuple of size 2

The argument to the function 'rev' must be a sequence

The argument to the function 'head' must be a sequence

The argument to the function 'last' must be a sequence

The argument to the function 'tail' must be a sequence

The argument to the function 'front' must be a sequence

The argument to ‘first’ must be a tuple of size 2

The argument to ‘second’ must be a tuple of size 2

Argument to '#' must be either a set or a sequence

Subscript to a sequence must be a number

Expression used as a function must evaluate to a set of a tuple

Expression used as a function must evaluate to a tuple

The tuple must be a binary relation

Parameter types do not match function parameters

Expression for ***o*** must evaluate to a set

All expressions of a bag must be of the same type

Left-hand expression for ‘#’ must be a bag

Right-hand expression for ‘#’ must be the same type as contained in the bag

Expressions for bag union must be bags

Both sides of bag union must be the same type