FROM FORMAL SPECIFICATION TO FULL PROOF: A STEPWISE METHOD

by

Lavinia Burski

Submitted for the degree of
Doctor of Philosophy

Department of Computer Science
School of Mathematical and Computer Sciences
Heriot-Watt University

March 2016

Acronyms

ASM Abstract state machine.

CGa Core Grammatical aspect.

DRa Document Rhetorical aspect.

GPSa General Proof Skeleton aspect.

Gpsa General Proof Skeleton aspect.

GpsaOL General Proof Skeleton ordered list.

Hol-Z Hol-Z.

IEC International Electrotechnical Commission.

MathLang MathLang framework for mathematics.

PPZed Proof Power Z.

SIL Safety Integrity Levels.

SMT Satisfiability Modulo Theories.

TSa Text and Symbol aspect.

UML Unified Modeling Language.

UTP Unifying theories of programming.

ZCGa Z Core Grammatical aspect.

ZDRa Z Document Rhetorical aspect.

ZMathLang MathLang framework for Z specifications.

Glossary

computerisation The process of putting a document in a computer format.

formal methods Mathermatically rigorous techniques and tools for the specification, design and verification of software and hardware systems.

formalisation The process of extracting the essence of the knowledge contained in a document and providing it in a complete, correct and unambiguous format.

halfbaked proof The automatically filled in skeleton also known as the Half-Baked Proof.

partial correctness A total correctness specification [P] C [Q] is true if and only if, whenever C is executed in a state satisfying P and if the execution of C terminates, then the state in which Cs execution terminates satisfies Q.

semi-formal specification A specification which is partially formal, meaning it has a mix of natural language and formal parts.

total correctness A total correctness specification [P] C [Q] is true if and only if, whenever C is executed in a state satisfying P, then the execution of C terminates, after C terminates Q holds.

Chapter 1

Overview of ZMathLang

Using the methodology of MathLang for mathematics (section ??), I have created and implemented a step by step way of translating Z specifications into theorem provers with additional checks for correctness along the way. This translation consists of one large framework (executed by a user interface) with many smaller tools to assist the translation. Not only is the translation useful for a novice to translate a formal specification into a theorem prover but it also creates other diagrams and graphs to help with the analysis of a formal system specification.

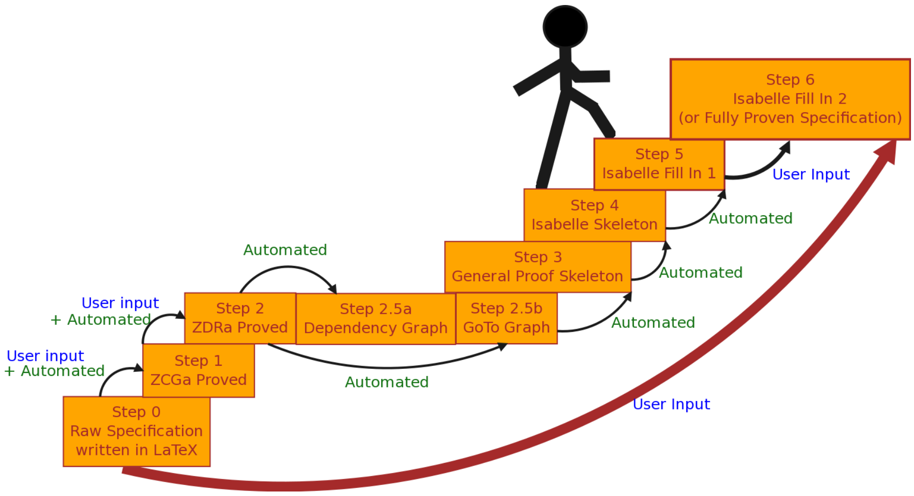


Figure 1.1: The steps required to obtain a full proof from a raw specification.

The framework is targeted at beginners in theorem proving. The users should have some idea of formal specifications but have no or little knowledge of the targetted theorem prover. Figure 1.1 shows the outline of the framework. The higher the user goes up the steps the more rigorous the checks for correctness. Step 1 and step 2 are interchangable and can be done in any order. However they both must be completed before moving up to step 3. Step 6 is the highest level of rigour and checks for full correctness in a theorem prover [1]. For this thesis I have chose to translate Z specifications into Isabelle, however this framework is an outline for any formal specification into any theorem prover which could done in the future.

The user doesn't need to go all the way to the top to check for correctness, one advantage of breaking up the translation is that the user gets some level of rigour and can be satisfied with some level of correctness along the way. However the main advantage of breaking up the translation is that the level of expertise needed to check for the correctness of a system specification can be done by someone who is not a theorem prover expert. This tool could also aid users in learning theorem proving as it translates their specification and thus they have examples of the syntx used in their theorem prover for their specification.

The arrows in figure 1.1 represent the amount of expertise needed for each step. In the last step, the arrow is slightly thicker as perhaps some theorem prover knowledge would be needed to complete the proofs. However these arrows are still small in comparison to the red thick arrow which represents translating the specification all in one go.

The framework breaks the translation into 6 steps, most of which are partially or fully automated. These are:

- Step 0: Raw LaTeX Z Specification. Start
- Step 1: Check for Core Grammatical correctness (ZCGa). User Input + Automated
- Step 2: Check for Document Rhetorical correctness (ZDRa). User Input + Automated
- Step 3: Generate a General Proof Skeleton (GPSa). Automated

[1]Full correctness in reference to completing sanity checks of the specification. Full correctness can be variable depending on the users choice. This is further discussed in chapter ??.

- Step 4: Generate an Isabelle Skeleton. Automated
- Step 5: Fill in the Isabelle Skeleton. Automated
- Step 6: Prove existing lemmas and add more safety properties if needed. User Input

1.1 How far does the automation go?

Figure 1.2 shows a diagram showing how far one can automate a specification using automated MathLang framework for Z specifications (ZMathLang) and Isabelle tools. ZMathLang is a toolset which assists the user in translating and proving a specification (going from left to right). There are also other automating tools within Isabelle which also assist the user with proving specifications (going from right to left) in the diagram.

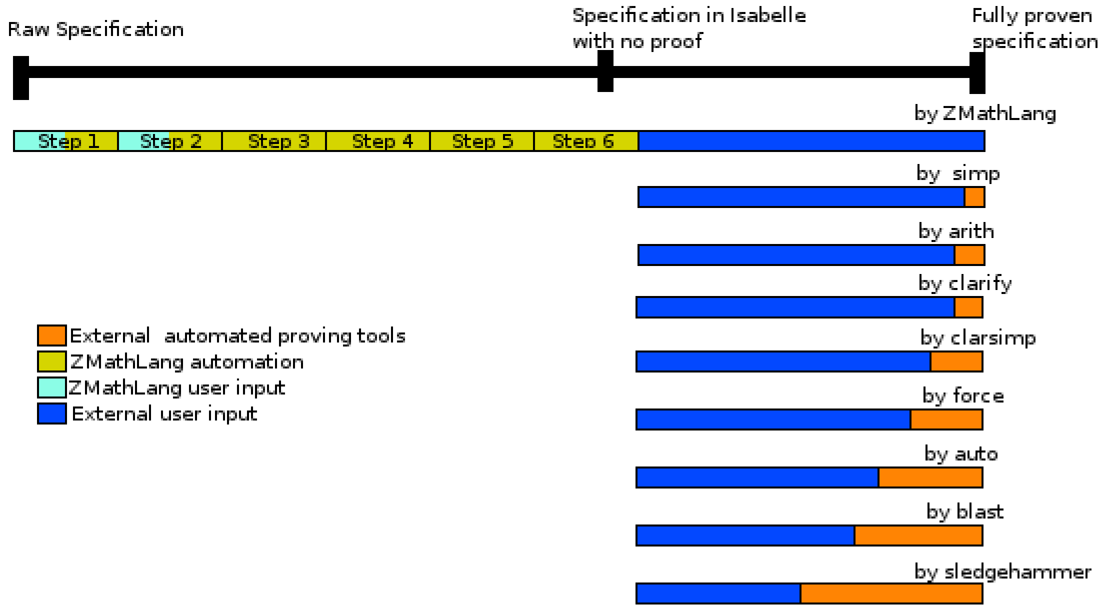


Figure 1.2: How far can one automate a specification proof.

In figure 1.2 we show how far the user can get with automation and how much work is still needed to get the full proof. ZMathLang requires user input for the first two steps (Z Core Grammatical aspect (ZCGa) and Z Document Rhetorical aspect (ZDRa)) however the rest up to step 6 is automated.

The black line shows the path going from a raw specification to a fully proven specification with a milestone in the middle, which signifies when a specification has been translated into Isabelle syntax but has no properties or proof. ZMathLang takes the user a little past this milestone as the toolset also generates properties to check the specification for consistency (see section **??**). These properties are added to the specification during step 3 and continued throughout the translation. It is important to note that the ZMathLang toolkit adds these properties to the translation but does not prove them. That is why the rest of the ZMathLang path may require external user input (dark blue) to complete the path. However, the ZMathLang toolkit does assist the user in the translation past the halfway milestone on the diagram.

We have created the ZMathLang toolkit which assist the user from the specification to full proof however there is also ongoing research on proving properties from the theorem prover end. Figure 1.2 shows the amount of proving techniques each automation holds. We have highlighted that ZMathLang only gets the user so far in their proof however they are free to use external automated theorem provers in completing their specification proof if they so wish.

Even external automated theorem provers have their limitations. For example, the user can use the Isabelle tool '*sledgehammer*' to assist in solving proofs, but not all can be solved by this technique. The sledgehammer documentation advises to call '*auto*' or '*safe*' followed by '*simp_all*' before invoking sledgehammer. Depending on the complexity of ones proof, these sometimes may prove the users properties on their own, other times it may not and the user will still need to invoke sledgehammer to reach their goal. Sledgehammer itself is a tool that applies Satisfiability Modulo Theories (SMT) solvers on the current goal e.g. Vampire[67], SPASS [25] and E [70]. We use sledghammer as a collective, to describe all the SMT solvers it covers [9].

Other automated methods include:

- **simp:** simplifies the current goal using term rewriting.
- **arith:** automatically solves linear arithmetic problems.
- **clarify:** like '*auto*' but less aggressive.

- **clarsimp:** a combination of '*clarify*' and '*simp*'.
- **force:** like '*auto*' but only applied to the first goal.
- **auto:** applies automated tools to look for a solution.
- **blast:** a powerful first-order prover. [24]

All these automated tools get increasingly more complex and cover more properties, e.g *clarsimp* covers more proving techniques then *simp* and *blast* covers more proving techniques than *auto* etc. With these tools, one can prove certain properties about their theorem. However, there still doesn't exist an automated proving tool which covers **all** proving techniques. Therefore some user input will be required for more complex proofs.

1.2 Overview of ZMathLang step by step

This section gives an overview of each indvidual step in the ZMathLang toolset.

1.2.1 Step 0- The raw LaTeX file

The first step requires the user to write or have a formal specification they wish to check for correctness. This specification can be fully written in Z or partially written in Z (thus a specification written in English on it's way to becoming formalised in Z). The specification should be written in LaTeX format and can be a mix of natural language and Z syntax. An example of a specification written in the Z notation can be seen in figure 1.3.

$[NAME, DATE]$

BirthdayBook

$known : \mathbb{P}\, NAME$

$birthday : NAME \pfun DATE$

$known = \mathrm{dom}\, birthday$

InitBirthdayBook

$BirthdayBook'$

$known' = \{\}$

AddBirthday

$\Delta BirthdayBook$

$name? : NAME$

$date? : DATE$

Figure 1.3: Example of a partial Z specification.

1.2.2 Step 1- The Core Grammatical aspect for Z

The next step in figure 1.1 shows that the specification should be ZCGa proved. Although this step is interchangable with step 2 (ZDRa) it is shown as step 2 on the diagram for convinience. In this step the user annoates their document which they have obtained in step 0 with 7 grammatical categories and then checks these for correctness. Figure 1.1 shows this step is achieved by user input and automation. The user input of this step is the annotations and the automation is the ZCGa checker. This automatically produces a document labeled with the various categories in differt colours and can help identify grammar types to other members in the systems project team. A ZCGa annotated specification is shown in figure 1.4. The ZCGa is further explained in chapter **??**.

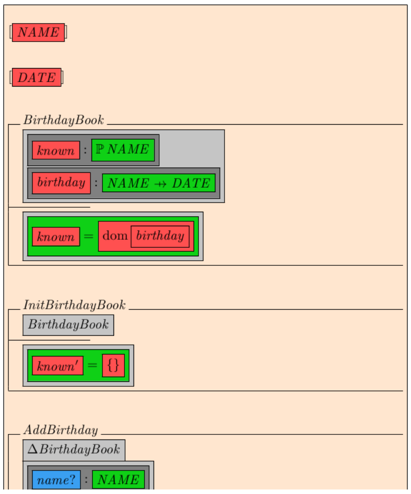


Figure 1.4: Example of a ZCGa annotated specification.

1.2.3 Step 2- The document Rhetorical aspect for Z

The ZDRa step, shown as step 2 in figure 1.1, comes before or after the ZCGa step. Similarly to the ZCGa step, the user annotates their document from step 0 or step 1 with ZDRa instances and relationships. This chunks parts of the specification together and allows the user to describe the relationship between these chunks. The annotation is the user input part of this step and the automation is the ZDRa checker which checks if there are any loops in the reasoning and gives warnings if the specification still needs to be totalised. Once the user has annotated this document and compiled it the outputing result shows the specification divided into chunks and arrows showing the relations between the chunks. An example of a Z specification annotated in ZDRa is shown in figure 1.5. The ZDRa is explained further in chapter ??.

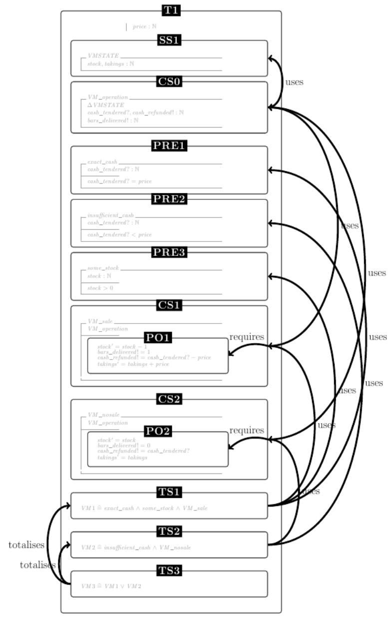


Figure 1.5: Example of a ZDRa annotated specification.

The ZDRa automatically produces a dependency and a goto graph (section **??**), these are shown as 2.5a and 2.5b respectively in figure 1.1. The loops in reasoning are checked in both the dependecy graph and goto graph. An example of a goto graph is shown in figure 1.6.

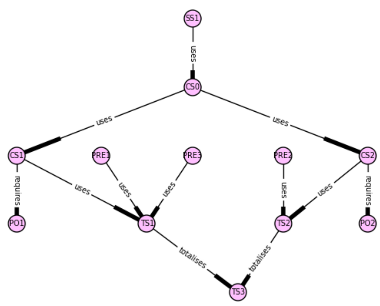


Figure 1.6: Example of an automatically generated goto graph.

1.2.4 Step 3- The General Proof skeleton

The following step is an automatically generated General Proof Skeleton aspect (Gpsa). This document is automated using the goto graph which is generated from the ZDRa annotated LaTeX specification. It uses the goto graph to describe in which logical order to input the specification into any theorem prover. At this stage it also adds simple proof obligations to check for the consitancy of the specification i.e. the specification is consistant throughout. An example of a general proof skeleton is shown in figure 1.7. The Gpsa is further described in section ??.

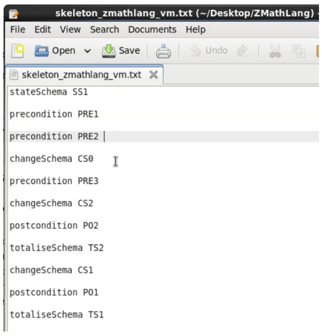
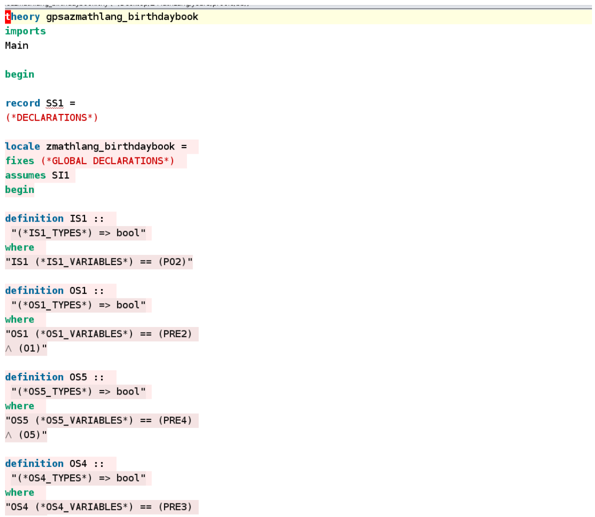


Figure 1.7: Example of a general proof skeleton.

1.2.5 Step 4- The Z specification written as an Isabelle Skeleton

Using the Gpsa in step 3, the instances are then translated into an Isabelle skeleton in step 4. That is, the instances of the specification are translated into Isabelle syntax using definitions, lemma's, theorys etc to produce a .thy file. This step is fully automated and thus a user with no Isabelle experience can still get to this stage. An example of a Z specification skeleton written in Isabelle is shown in figure 1.8. Details of how this translation is conducted is described in chapter ?? of this thesis.



```
theory gpsazmathlang_birthdaybook
imports
Main

begin

record SS1 =
(*DECLARATIONS*)

locale zmathlang_birthdaybook =
fixes (*GLOBAL DECLARATIONS*)
assumes SI1
begin

definition IS1 ::
 "(*IS1_TYPES*) => bool"
where
"IS1 (*IS1_VARIABLES*) == (PO2)"

definition OS1 ::
 "(*OS1_TYPES*) => bool"
where
"OS1 (*OS1_VARIABLES*) == (PRE2)
∧ (O1)"

definition OS5 ::
 "(*OS5_TYPES*) => bool"
where
"OS5 (*OS5_VARIABLES*) == (PRE4)
∧ (O5)"

definition OS4 ::
 "(*OS4_TYPES*) => bool"
where
"OS4 (*OS4_VARIABLES*) == (PRE3)
```

Figure 1.8: Example of an Isabelle skeleton.

1.2.6 Step 5- The Z specification written as in Isabelle Syntax

Step 5 is also automated, using the ZCGa annotated document produced in step 1 and the Isabelle skeleton produced in step 4. This part of the framework fills in the details from the specification using all the declarations, expressions, definition etc in Isabelle syntax. Since the translation can also be done on semi-formal specifications and incomplete formal specification there may be some information missing in the ZCGa such as an expression or a definition. Note the lemmas from the proof obligations created in step 3 will also be filled in, however the actual proofs for these will not and they will be followed by the command '`sorry`' to artificially complete the proof. An example of a filled in isabelle skeleton is shown in figure 1.9.

```
theory 5
imports
Main
begin

record VMSTATE =
STOCK :: nat
TAKINGS :: nat

locale zmathlang_vm =
fixes stock :: "nat"
and takings :: "nat"
and price :: "nat"
begin

definition exact_cash ::
 "nat => bool"
where
"exact_cash cash_tendered = (cash_tendered = price) "

definition insufficient_cash ::
 "nat => bool"
where
"insufficient_cash cash_tendered = (cash_tendered < price) "

definition VM_operation ::
```

Figure 1.9: Example of an Isabelle skeleton automatically filled in.

If there is no ZCGa information to fill in the Isabelle skeleton will not change. Further information on the translation is described in section **??** of this thesis.

1.2.7 Step 6- A fully proven Z specification

The final step in the ZMathLang framework (top of the stairs from figure 1.1), is to fill in the Isabelle file from step 5. This final step is represented by a slightly thicker arrow in figure 1.1 compared with the others as the user may need to have some little theorem prover knowledge to prove properties. Also if there is some missing information such as missing expressions and definitions the user must fill these out as well in order to have a fully proven specification. However this may be slightly easier then writing the specification from scratch as the user would already have examples of other instances in their Isabelle syntax form. More details on this last step is described in section **??** of this thesis.

1.3 Procedures and products within ZMathLang

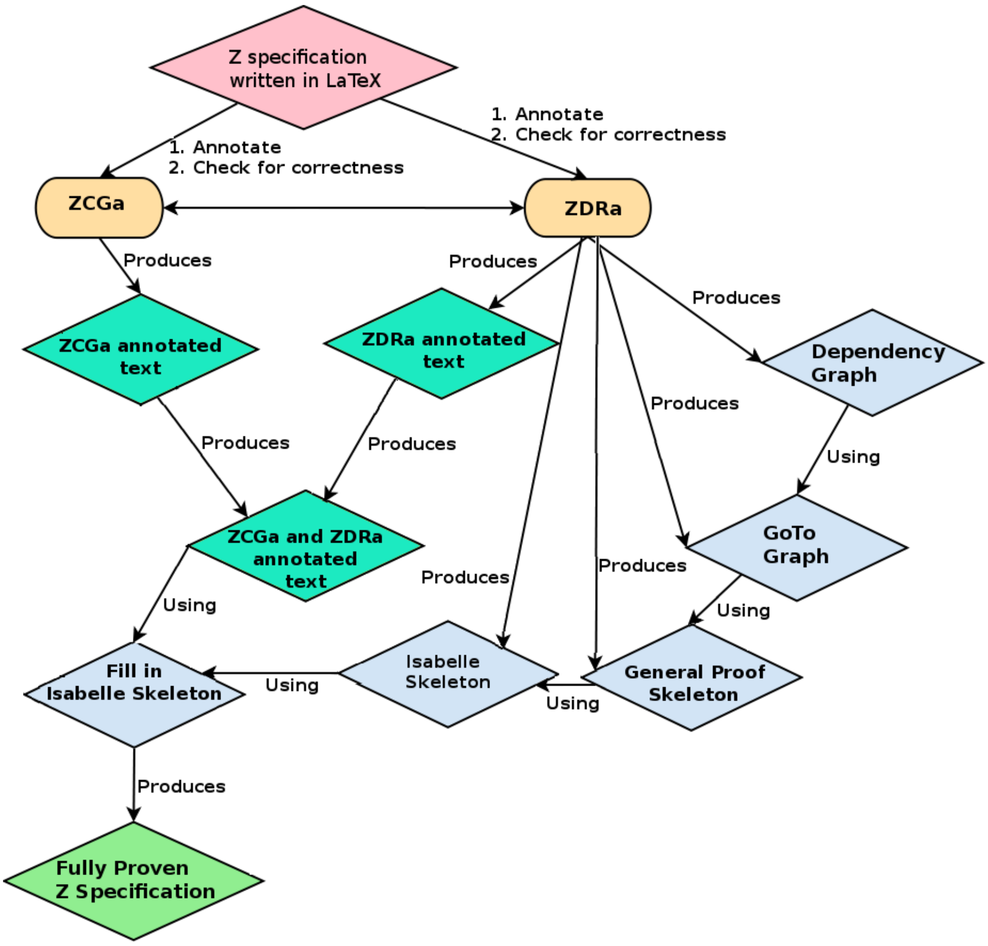


Figure 1.10: Flow chart of ZMathLang.

Figure 1.10 shows a flow chart describing the documents produced when using the framework and which documents are produced automatically, semi-automatically and totally by the user. Products which are created by full automation are diamonds in blue. Diamonds in green are produced by user input and products shown in aqua diamonds are partial automated.

The pink diamond is the starting point for all users. The orange ovals describe procedures of the ZCGa and ZDRa. The ZCGa procedure requires user input and automation to produce a 'ZCGa annotated text'. The ZDRa procedure also requires user input and automation to produce a 'ZDRa annotated text'. Both the ZCGa and ZDRa procedures done together produce a 'ZCGa and ZDRa annotated text'. After completing the ZDRa procedure a 'dependency graph' is automatically generated, which can then in turn generate a 'GoTo graph' which in turn can create a general proof skeleton. From the 'general proof skeleton' we can then create an 'Isabelle skeleton' which can be filled in using information from the 'ZCGa and ZDRa annotated text'. Using the 'Filled in Isabelle skeleton' the user needs to fill in the missing information and/or complete the proofs in order to obtain a 'fully proven Z specification'.

1.4 The ZMathLang LaTeX Package

The ZMathLang LaTeX package (shown in appendix **??**) was implemented to allow the user to label their Z specification document in ZCGa and ZDRa annotations. Coloured boxes will then appear around the grammatical categories when the new ZCGa annotated document is compiled with `pdflatex`. Instances and labelled arrows showing the relations are also displayed when annotated with ZDRa and compiled with `pdflatex`.

1.4.1 Overview

The ZMathLang style file invokes the following packages:

- tcolorbox - Used to draw colours around individual grammatical categories with a black outline for the ZCGa.
- tikz - Used to identify the instances as nodes so the arrows can join from one nodes to another.
- varwidth - Used to chunk each instance as a single entity.
- zed - Used to draw Z specification schemas, freetypes, axiomatic definitions in the zed environment.
- xcolor - Used to define specific colours and gives a wider range of colours compared to the standard.

After invoking the packages we define the colours which are used in the outputting pdf result. We use the same colours as the original MathLang framework for mathematics (MathLang) framework for the grammatical categories which are the same (sets, terms, expressions, declarations, context and definitions) and choose a different colour for the weak type 'specification' as this hasn't been used in the original MathLang framework.

```
\definecolor{term}{HTML}{3A9FF1}
```

Figure 1.11: Part of the syntax to define the colours for ZCGa in the ZMathLang LaTeX file.

The command `\definecolor{*NameOfZCGaType*}{HTML}{*ColourInHtml*}` is used to define a colour for each grammatical category (shown in figure 1.11). Where `*NameOfZCGaType*` is the name of the category e.g. definition, term, set etc and `*ColourInHtml*` is the HTML number for the colour. For example the colour for term in the original ZMathLang is *lightblue* which in HTML format is *3A9FF1*. Therefore we define the colour for '*term*' as *3A9FF1*.

1.4.2 LaTeX commands to identify ZDRa Instances

The ZDRa section of the LaTeX file provides three new commands: `\draschema`, `\draline` and `\dratheory`. The `\dratheory` annotation is for the entire specification which contins all the intances and relations. The `\draschema` command is to annotate the instances which are entire zed schemas, this command should go before any `\begin{schema}` or `\begin{zed}` command.

`\draline{X}{\draschema{Y}{someContext}}`	Incorrect
`\draschema{Y}{\draline{X}{someContext}}`	Correct

The `\draline` annotation is to annotate any instance that is a line of text which contains plain text or ZCGa annotated text. But does not include any ZDRa annotated text. For example in figure 1.12 the `\draline{PRE1}` annotation is embedded in the `\draline{CS1}{` which will not compile. Therefore the correct way this schema is labelled is shown in figure 1.13 where the `\draline{PRE1}` annotation is embedded in the `\draschema{CS1}` annotation.

```
\draline{CS1}{
\begin{schema}{B}
\Delta A
\where
\draline{PRE1}{a<b}
\end{schema}
}
```

Figure 1.12: Incorrect annotating of ZDRa.

```
\draschema{CS1}{
\begin{schema}{B}
\Delta A
\where
\draline{PRE1}{a<b}
\end{schema}
}
```

Figure 1.13: Correct annotating of ZDRa.

It is important to note this embedding order as by annotating a chunk of specification using `\draline` keeps everything inside as the LaTeX '*math mode*'. Since the annotation `\draschema` is outside the zed commands (eg `\begin{schema}`) it does not convert the content into '*math mode*' but the zed commands do.

```
\newcommand\draschema[2]{%
\begin{tcolorbox}[colback=white, enhanced, overlay,.
remember as=#1, finish={\node[] at (frame.north) {
\LARGE
\bfseries
\colorbox{Black}{\color{White}#1}};}]
{\color{Gray}\begin{varwidth}
{\dimexpr\linewidth-2\fboxsep}#2\end{varwidth}}
\end{tcolorbox}
}
```

Figure 1.14: The syntax to define a ZDRa schema instance in the ZMathLang LaTeX file.

The new command we are defining for `\draschema` is shown in figure 1.14. The commands for defining `\dratheory` and `\draline` are similar as the `draschema` definition. The command takes two arguments, the first argument will be the name of the instance (e.g SS1, IS4, CS2 etc) and the second argument is the instance itself. Any text within the instance will then become grey so it looks faded as we are only interested in the instance itself and not the context at this point. The background of the box is white with a black outline. We then use the first argument to name the instance and it becomes a node. The name of the instance is also printed in black over the instance itself.

1.4.3 LaTeX commands to identify ZDRa Relations

There are 5 new commands to define the relations for the ZDRa, these are *initialOf*, *uses*, *totalises*, *requires* and *requires*. Information on these relations are described in chapter **??**, however this section of the thesis describes how the annotations have been implemented in the ZMathLang LaTeX package.

```
\newcommand\uses[2]{
\begin{tikzpicture}[overlay,remember picture
,line width=1mm,draw=black!75!black, bend angle=90]
\draw[->] (#1.east) to[bend right] node[right, Black]
{\LARGE{uses}} (#2.east);
\end{tikzpicture}
}
```

Figure 1.15: The syntax to define a ZDRa schema relation in the ZMathLang LaTeX file.

Figure 1.15 shows how the command `uses` has been implemented. The command takes 2 arguments (the should be 2 instances which have been previously annotated) and draws an arrow going from the first instance to the second one. The arrow bend angle is at 90, the arrow width is at 1mm and the arrow goes from the east part of the first instance to the east part of the second instance. The word **uses** is written next to the arrow. All the other relation commands are written in a similar way however the direction of the arrows differ and some arrows bend to the left whilst others bend to the right. The bending of the arrows has been implemented at random so that the compiled document has arrows showing on both sides of the theory and are not overlapping too much.

1.4.4 LaTeX commands to identify ZCGa grammatical types

The ZCGa part of the LaTeX file package uses the colours previously defined in the style file. To define each of the grammatical types we use the `fcolorbox` command. This creates a black outline and a coloured background for each of the grammatical categories.

```
\newcommand\declaration[1]{
\fcolorbox{Black}{declaration}{$#1$}
}

\renewcommand\set[1]{
\fcolorbox{Black}{set}{$#1$}
}
```

Figure 1.16: The syntax to define a ZCGa grammatical categories.

Figure 1.16 shows the commands to define the coloured boxes for *declaration* and *set*. As set is already defined in the mathematical LaTeX library, we renew the command. The command takes one argument (the text the user which to annotate), changes it to mathmode and draws the box around it. All the grammatical categories are defined in the same way, each with their own background colour. The only exception is the grammatical category of *specification* as this command does not convert the specification into mathmode.

1.5 Conclusion

In total there are 6 steps in order to translate a Z specification into the theorem prover Isabelle. These steps have been designed so that the system engineer/system designer of the project could use them. Each of these steps assist the user in understanding the specification, and some steps even produce documents, graphs and charts in order to analyse the specification. These products also allow others in the development team to understand the system better such as clients, stakeholders, developers etc. The majority of the steps are fully automated whilst some a little user input. Each step checks for some form of correctness and becomes more and more rigorous each step the user takes towards step 6. The next chapter begins to describe step 1 (ZCGa) in more detail.

Bibliography

[1] J.-R. Abrial. Event Based Sequential Program Development: Application to Constructing a Pointer Program. In K. Araki, S. Gnesi, and D. Mandrioli, editors, *FME*, volume 2805 of *Lecture Notes in Computer Science*, pages 51–74. Springer, 2003.

[2] J.-R. Abrial. Formal methods in industry: achievements, problems, future. *Software Engineering, International Conference on*, 0:761–768, 2006.

[3] M. Adams. Proof auditing formalised mathematics. *Journal of Formalized Reasoning*, 9(1):3–32, 2016.

[4] A. Álvarez. *Automatic Track Gauge Changeover for Trains in Spain.* Vía Libre monographs. Vía Libre, 2010.

[5] A. W. Appel. Foundational Proof-Carrying Code. In *LICS*, pages 247–256, 2001.

[6] R. Arthan. Proof Power. `http://www.lemma-one.com/ProofPower/index/`, February 2011.

[7] H. P. Barendregt. Lambda Calculi with Types. In *Handbook of Logic in Computer Science*, volume 2. Oxford University Press, 1991. http://citeseer.ist.psu.edu/barendregt92lambda.htmlElectronic Edition.

[8] B. Beckert. An Example for Specification in Z: Steam Boiler Control. Universitat Koblenz-Landau, Lecture Slides, 2004.

[9] J. C. Blanchette. *Hammering Away, A user's guide to Sledgehammer for Isabelle/HOL*. Institut fur Informatik, Technische Universitat Munchen, May 2015.

[10] E. Borger and R. F. Stark. *Abstract State Machines: A Method for High-Level System Design and Analysis*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2003.

[11] N. Bourbaki. *General topology. Chapters 1-4*. Elements of mathematics. Springer-Verlag, Berlin, Heidelberg, Paris, 1989. Trad. de : Topologie gnrale chapitres 1-4.

[12] J. Bowen. Formal Methods Wiki, Z notation. `http://formalmethods.wikia.com/wiki/Z_notation`, July 2014.

[13] A. D. Brucker, H. Hiss, and B. Wolff. HOL-Z 2.0: A Proof Environment for Z-Specifications. *Journal of Universal Computer Science*, 9(2):152–172, feb 2003.

[14] L. Burski. Zmathlang. `http://www.macs.hw.ac.uk/~lb89/zmathlang/`, Jan 2016.

[15] L. Burski. ZMathLang Website. `http://www.macs.hw.ac.uk/~lb89/zmathlang/examples`, June 2016.

[16] R. W. Butler. An introduction to requirements capture using PVS: Specification of a simple autopilot. NASA Technical Memorandum 110255, NASA Langley Research Center, Hampton, VA, May 1996.

[17] R. W. Butler. What is Formal Methods. `http://shemesh.larc.nasa.gov/fm/fm-what.html`, March 2001.

[18] W. Chantatub. *The Integration of Software Specification Verification and Testing Techniques with Software Requirements and Design Processes*. PhD thesis, University of Sheffield, 1995.

[19] Clearsy Systems Engineering. B Methode. `http://www.methode-b.com/en/`, 2013.

[20] J. Coleman, C. Jones, I. Oliver, A. Romanovsky, and E. Troubitsyna. RODIN (rigorous open development environment for complex systems). In *EDCC-5, Budapest, Supplementary Volume*, pages 23–26, Apr. 2005.

[21] I. E. Commission. IEC 61508 Functional Safety of Electrical/Electronic/Programmable Electronic Safety-related Systems. Technical report, International Electrotechnical Commission, 2010.

[22] E. Currie. *The Essence of Z.* Prentice-Hall Essence of Computing Series. Prentice Hall Europe, 1999.

[23] H. Curry. Functionality in combinatorial logic. In *Proceedings of National Academy of Sciences*, volume 20, pages 584–590, 1934.

[24] C.Weidenbach, D.Dimova, A.Fietzke, R.Kumar, M.Suda, and P. Wischnewski. Isabelle cheat sheet. `http://www.phil.cmu.edu/~avigad/formal/FormalCheatSheet.pdf`.

[25] C.Weidenbach, D.Dimova, A.Fietzke, R.Kumar, M.Suda, and P. Wischnewski. Spass. `http://www.spass-prover.org/publications/spass.pdf`.

[26] N. de Bruijn. The mathematical vernacular, a language for mathematics with typed set. *In Workshop on Programming Logic*, 1987.

[27] L. De Moura and N. Bjørner. Satisfiability Modulo Theories: Introduction and Applications. *Commun. ACM*, 54(9):69–77, Sept. 2011.

[28] D. Fellar, F. Kamareddine, and L. Burski. Using MathLang to Check the Correctness of Specifications in Object-Z. In E. Venturino, H. M. Srivastava, M. Resch, V. Gupta, and V. Singh, editors, *In Modern Mathematical Methods and High Performance Computing in Science and Technology*, Ghaziabad, India, 2016. M3HPCST, Springer Proceedings in Mathematics and Statistics.

[29] D. Feller. Using MathLang to check the correctness of specification in Object-Z. Master Thesis Report, 2015.

[30] Formal Methods Europe, L-H Eriksson. Formal methods europe. `http://www.fmeurope.org/?page_id=2`, May 2016.

[31] S. Fraser and R. Banach. Configurable Proof Obligations in the Frog Toolkit. In *Fifth IEEE International Conference on Software Engineering and Formal Methods (SEFM 2007), 10-14 September 2007, London, England, UK*, pages 361–370. IEEE Computer Society, 2007.

[32] J. Groote, A. Osaiweran, and Wesselius2. Benefits of Applying Formal Methods to Industrial Control Software. Technical report, Eindhoven University of Technology, 2011.

[33] S. L. Hantler and J. C. King. An Introduction to Proving the Correctness of Programs. *ACM Comput. Surv.*, 8(3):331–353, Sept. 1976.

[34] E. C. R. Hehner. Specifications, Programs, and Total Correctness. *Sci. Comput. Program.*, 34(3):191–205, 1999.

[35] A. Ireland. Rigorous Methods for Software Engineering, High Integrity Software Intensive Systems. Heriot Watt Universtiy, MACS, Lecture Slides.

[36] F. Kamareddine and J.B.Wells. A research proposal to UK funding body. Formath, 2000.

[37] F. Kamareddine, R. Lamar, M. Maarek, and J. B. Wells. Restoring Natural Language as a Computerised Mathematics Input Method. In M. Kauers, M. Kerber, R. Miner, and W. Windsteiger, editors, *Calculemus/MKM*, volume 4573 of *Lecture Notes in Computer Science*, pages 280–295. Springer, 2007.

[38] F. Kamareddine, M. Maarek, K. Retel, and J. B. Wells. Gradual computerisation/formalisation of mathematical texts into Mizar. In *From Insight to Proof: Festschrift in Honour of Andrzej Trybulec*, pages 81–95. Springer-Verlag, 2007.

[39] F. Kamareddine, M. Maarek, and J. B. Wells. Toward an Object-Oriented Structure for Mathematical Text. In M. Kohlhase, editor, *MKM*, volume 3863 of *Lecture Notes in Computer Science*, pages 217–233. Springer, 2005.

[40] F. Kamareddine and R. Nederpelt. A refinement of de Bruijn's formal language of mathematics. *Logic, Language and Information*, 13(3):287–340, 2004.

[41] F. Kamareddine, J. B. Wells, and C. Zengler. Computerising mathematical texts in MathLang. Technical report, Heriot-Watt University, 2008.

[42] Khosrow-Pour and Mehdi, editors. *Encyclopedia of Information Science and Technology*. IGI Global,, 2 edition.

[43] S. King, J. Hammond, R. Chapman, and A. Pryor. Is Proof More Cost-Effective Than Testing? *IEEE Trans. Software Eng.*, 26(8):675–686, 2000.

[44] Kolyang, T. Santen, and B. Wolff. *Theorem Proving in Higher Order Logics: 9th International Conference, TPHOLs'96 Turku, Finland, August 26–30, 1996 Proceedings*, chapter A structure preserving encoding of Z in isabelle/HOL, pages 283–298. Springer Berlin Heidelberg, Berlin, Heidelberg, 1996.

[45] Kolyang, T. Santen, B. Wolff, R. Chaussee, I. Gmbh, and D.-S. Augustin. Towards a Structure Preserving Encoding of Z in HOL, 1986.

[46] A. Krauss. Defining Recursive Functions in Isabelle/HOL , 2008.

[47] R. Lamar. The MathLang Formalisation Path into Isabelle – A Second-Year report, 2003.

[48] R. Lamar. *A Partial Translation Path from MathLang to Isabelle*. PhD thesis, Heriot-Watt University, 2011.

[49] R. Lamar, F. Kamareddine, and J. B. Wells. MathLang Translation to Isabelle Syntax. In J. Carette, L. Dixon, C. S. Coen, and S. M. Watt, editors, *Calculemus/MKM*, volume 5625 of *Lecture Notes in Computer Science*, pages 373–388. Springer, 2009.

[50] P. G. Larsen, N. Battle, M. Ferreira, J. Fitzgerald, K. Lausdahl, and M. Verhoef. The overture initiative integrating tools for vdm. *SIGSOFT Softw. Eng. Notes*, 35(1):1–6, Jan. 2010.

[51] I. Lee, J. Y.-T. Leung, and S. H. Son. *Handbook of Real-Time and Embedded Systems*. Chapman & Hall/CRC, 1 edition, 2007.

[52] K. R. M. Leino. Dafny: An Automatic Program Verifier for Functional Correctness. In E. M. Clarke and A. Voronkov, editors, *LPAR (Dakar)*, Lecture Notes in Computer Science, pages 348–370. Springer, 2010.

[53] M. Lindgren, C. Norstrm, A. Wall, and R. Land. Importance of Software Architecture during Release Planning. In *WICSA*, pages 253–256. IEEE Computer Society, 2008.

[54] I. E. U. Ltd and H. . S. Laboratory. A methodology for the assignment of safety integrity levels (SILs) to safety-related control functions implemented by safety-related electrical, electronic and programmable electronic control systems of machines. Standard, Health and Safety Executive (HSE), Mar. 2004.

[55] M. Maarek. Mathematical documents faithfully computerised:the grammatical and text & symbol aspects of the MathLang framework, First Year Report, 2003.

[56] M. Maarek. *Mathematical documents faithfully computerised: the grammatical and test & symbol aspects of the MathLang Framework*. PhD thesis, Heriot-Watt University, 2007.

[57] M. Mahajan. Proof Carrying Code. *INFOCOMP Journal of Computer Science*, 6(4):01–06, 2007.

[58] M. Mihaylova. ZMathLang User Interface Internship Report. Internship Report, 2015.

[59] M. Mihaylova. ZMathLang User Interface User Manual. Intern User Manual, 2015.

[60] G. C. Necula and P. L. 0001. Safe, Untrusted Agents Using Proof-Carrying Code. In G. Vigna, editor, *Mobile Agents and Security*, volume 1419 of *Lecture Notes in Computer Science*, pages 61–91. Springer, 1998.

[61] I. C. Office. International Electrotechnical Commission. `http://www.iec.ch/`, July 2016.

[62] S. Owre, S. Rajan, J. Rushby, N. Shankar, and M. Srivas. PVS: combining specification, proof checking, and model checking. In R. Alur and T. A. Henzinger, editors, *Computer-Aided Verification, CAV '96*, number 1102 in Lecture Notes in Computer Science, pages 411–414, New Brunswick, NJ, July/August 1996. Springer-Verlag.

[63] R. L. Page. Engineering Software Correctness. *J. Funct. Program.*, 17(6):675–686, 2007.

[64] B. C. Pierce. *Types and Programming Languages.* MIT Press, Cambridge, MA, USA, 2002.

[65] W. R. Plugge and M. N. Perry. American Airlines' "Sabre" Electronic Reservations System. In *Papers Presented at the May 9-11, 1961, Western Joint IRE-AIEE-ACM Computer Conference*, IRE-AIEE-ACM '61 (Western), pages 593–602, New York, NY, USA, 1961. ACM.

[66] K. Retel. *Gradual Computerisation and Verification of Mathematics: MathLang's Path into Mizar.* PhD thesis, Heriot-Watt University, 2009.

[67] A. Riazanov and A. Voronkov. The design and implementation of vampire. *Journal of AI Communications*, 15(2/3):91–110, 2002.

[68] G. Rossum. Python Reference Manual. Technical report, Python Software Foundation, Amsterdam, The Netherlands, The Netherlands, 1995.

[69] M. Saaltink and O. Canada. The Z/EVES 2.0 User's Guide, 1999.

[70] S. Schulz. E–a brainiac theorm prover. *Journal of AI Communications*, 15(2/3):111–126, 2002.

[71] J. M. Spivey. *The Z Notation: A Reference Manual*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1989.

[72] M. Spivey. Z Reference Card. `https://spivey.oriel.ox.ac.uk/mike/fuzz/refcard.pdf`. Accessed on November 2014.

[73] M. Spivey. Towards a Formal Semantics for the Z Notation. Technical Report PRG41, OUCL, October 1984.

[74] M. Spivey. The fuzz manual. *Computing Science Consultancy*, 34, 1992.

[75] S. Stepney. A tale of two proofs. In *BCS-FACS third Northern formal methods workshop, Ilkley*, 1998.

[76] I. UK. *Customer Information Control System (CICS) Application Programmer's Reference Manual*. White Plains, New York.

[77] University of Cambridge and Technische Universitat Munchen. Isabelle. `http://www.isabelle.in.tum.de`, May 2015.

[78] Z. Wen, H. Miao, and H. Zeng. Generating Proof Obligation to Verify Object-Z Specification. In *Proceedings of the International Conference on Software Engineering Advances (ICSEA 2006), October 28 - November 2, 2006, Papeete, Tahiti, French Polynesia*, page 38. IEEE Computer Society, 2006.

[79] A. Whitehead and B. Russell. *Principia Mathematica*. Number v. 2 in Principia Mathematica. University Press, 1912.

[80] J. Woodcock and A. Cavalcanti. A tutorial introduction to designs in unifying theories of programming. In *Integrated Formal Methods*, pages 40–66. Springer, 2004.

[81] J. Woodcock and J. Davies. *Using Z: Specification, Refinement, and Proof*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1996.

[82] C. Zengler. MathLang- Towards a Better Usability and Building the Path into Coq, First Year Report. Technical report, Heriot-Watt University, November 2008.