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

Abstract

Write the abstract here.

Acknowledgements

Contents

Chapter 1

Introduction

Industries developing high integrity software are always looking for ways to make their software safer. Safety Integrity Levels (SIL) are used to define a level of risk-reduction provided by a safety-function. The highest SIL which could be given to hardware or safety integrity system, as given by the International Electrotechnical Commission (IEC) standard, is a SIL4. A SIL4 has a probability of failure of between 0.0001 and 0.00001 [1], and although these probabilities are very low, they are non-zero, and the upper bound of 0.000001 suggests a failure every once every 1,000,000 times on average, the outcome of which can be catastrophic. Software testing usually takes place when the program or a prototype has been implemented. However by the time the product is fully implemented and errors are caught it is expensive to go back to the planning stage to find solutions to those bugs. Catching errors at an earlier stage of the project life cycle is more time and cost effective for the whole project team.

One way of detecting errors at an early stage of the project life cycle is by applying the use of formal methods at the design/specification stage of the project life cycle. The benefit of using formal methods is that they provide a means to symbolically examine the entire state space of a design and establish a correctness that is true for all possible inputs [4]. However due to the enormous complexity of real time systems they are rarely used. Formal methods come in different shapes and sizes; the Abstract state machine (ASM) theory is a state machine which operates on states or arbitrary data structures. The B-method [5] is a formal method for the development of program code from a specification in the ASM notation. Z [11] is a specification languages used for describing computer-based systems. Unified Modeling Language (UML) provides system architects with a consistent language for specifying, construing and documenting different components of systems. These are just a selection of various formal methods methods however there are a great deal more which are still applied to systems today to add a degree of safety to certain high integrity products.

Specifications models and verification may be done using different levels of rigour. Level 1 represents the use of mathematical logic to specify a system, level 2 uses a handwritten approach to proofs and level 3 is the most rigorous application of formal methods which uses theorem provers to undertake fully formal machine-checked proofs. Level 3 is clearly the most expensive level and is only practically worthwhile when the cost of making mistakes is extremely high.

The jump from Level 1 rigour to Level 3 rigour is very difficult, but in many cases worthwhile. The purpose of this thesis is to introduce an approach where the large jump is broken up in to multiple smaller jumps, allowing the level 3 of rigour to be more accessible and thus more widely used.

1.1 Motivations

In order to facilitate the computerisation process, this thesis proposes smaller computerisation steps which allow the translation (and hence the correctness checking) of specifications written in Z into a theorem prover such as ProofPower-Z [2] or Isabelle [13].

The reason to break the translation path into simple steps is because the original path is difficult and requires serious expertise in theorem proving and the translation from Z to a theorem prover. The approach approach described in this thesis, of mini-computerisation steps, allows different experts to collaborate on the various steps to build the final proof.

The list below provides the main motivations to the research of this thesis:

1.2 Contributions

A summary of contributions is given in the following points:

- ZMathLang's Z Core Grammatical aspect (ZCGa) has been created and implemented.
  - Weak types and weak typing rules have been thoroughly implemented.
  - A style file has been created to label a specification with ZCGa annotations. This style file also outputs coloured boxes around weak types in the specification so that the user can see the weak types in a clear manner.
  - A weak type checker, which reads the ZCGa annotations and checks they have been implemented.
  - Examples given for various specifications [1].
- ZMathLang's Z Document Rhetorical aspect (ZDRa) has been created and implemented. Dependency Graphs and Goto Graphs have been implemented to be automatically generated from the ZDRa annotated document.
  - Instance names and relations have been carefully realized and added to the LaTeX style file.
  - Relation rules have been outlined.
  - The ZDRa has been implemented to check for the document rhetorical correctness which outputs various warning and error messages.
  - Using directed graphs, dependency and GoTo graphs can be automatically generated from the implementations. Formal aspects of these graphs have also been highlighted.
  - Examples given for various specifications.

[1]The examples for various specifications can be found on `http://www.macs.hw.ac.uk/~lb89/zmathlang/examples`

- General Proof Skeleton aspect (GPSa) and Isabelle skeletons have been implemented so they are automatically generated from the ZDRa annotated document.
  - Using the Goto graph a general proof skeleton can be automatically created from the implementation.
  - Using the general proof skeleton an Isabelle skeleton and a filled in skeleton of the original specification can be automatically generated using the implementation.
  - A formal definition of the ZDRa, Dependency graph and GoTo graph has been given.
- A gradual computerisation path from Z specifications (BirthdayBook [11], Vending Machine [2] and all specifications in Curries [?]) have been documented. These are the first translations from raw Z specifications to complete proofs done using the MathLang framework for Z specifications (ZMathLang) framework. Out of these translations we get new dependency graphs and proof skeletons for the individual specifications.
  - Clear and concise translation paths from various Z specifications into ZCGa annotated and checked documents, ZDRa annotated and check documents.
  - Dependency graphs, GoTo Graphs and general proof skeletons are generated for all example specifications.
  - Isabelle skeletons automatically generated and filled in for all example specifications.
  - Safety property's and lemma's added to example specifications and proved.

1.3 Outline

In chapter **??** we begin describing the origins of MathLang framework for mathematics (MathLang), it's success and where it has been used so far. We then describe

Z specifications and the tools available for it so far.

In chapter **??**, we outline the basic idea of ZMathLang, how the original MathLang method has been adapted to perform with Z specifications and how this method is different to others previously described.

Chapter **??** provides more in depth details of the first contribution of this thesis. The weak types which have been created are presented as well as how they work together with weak typing rules. The categories which have been extracted from the weak types are presented and examples are given on how these categories correspond to Z specifications. Examples are given for all the weak types, and categories for Z specification. The weak type checker, which is implemented in `Python` [10], is thoroughly described and details of how the tool can be used is given.

Chapter **??** highlights another aspect which is a contribution of this thesis. An explanation of what rhetorical correctness for a specification is given. Instances and how they relate to each other are described as well as how a user can annotated these facts into a Z specification. Examples are given for all relations and instances and rules of what relations are allowed. An outline of the ZDRa checker is given, along with explanations of various error and warning messages. A general explanation of the dependency and GoTo graphs are also given.

In chapter **??** describes the different skeletons which can be automatically generated if the specification is ZDRa correct. A detail explanation of how a general proof skeleton can be created from the GoTo graph is given along with the algorithm which creates it. A summary is given of how the general proof sketch can be used to generate an Isabelle skeleton of the specification. Details of how the Isabelle skeleton can be filled in using the ZCGa annotated specification is also shown in this chapter. A demonstration of how we can use this filled in Isabelle skeleton to get a full proof is also described.

Chapter **??** gives formal definitions of the ZDRa correctness checker, dependency graphs and GoTo graphs. We prove various properties about the ZDRa using rules and definitions which have been given. We give examples of how each of these aspects can be represented in a formal manner. The algorithm which creates the dependency and GoTo graphs is given and explained.

In chapter **??** we give an overview on the interface design and we explain how one can use ZMathLang on Z specifications. Explanations of how to use each aspect is given via examples and screen shots. The tables of output messages which a user can receive are highlighted and explained.

Chapter **??** goes through one entire specification (vending machine) along the ZMathLang route. Each aspect is clearly highlighted and explained to the reader giving hints and tips along the way. Other examples are found in the appendix, however they are only taken along the ZMathLang route without any commentary.

In chapter **??** we consider 2 specification examples which have been proven in a theorem prover using a single step, and compare them with the same specification examples which have been proven in multiple steps using ZMathLang. We give a table of comparison explaining the amount of expertise required, type of input and lines of proofs and lemmas. We explain and compare the type of expertise required for each of the specification examples and how they compare against doing the proof in one step or multiple steps.

Finally, a conclusion is presented in chapter 2 which summarises the contributions made in this thesis. The limitations of this research and potential areas of future research are also discussed.

Chapter 2

Conclusion and Future Work

In this chapter we discuss the current development of ZMathLang and it's future works. We also conclude a comparisson between ZMathLang framework to other ststems. Finally in section we give add concluding thoughts to this thesis.

2.1 Achievements of this thesis

At the beginning of this thesis we described the motivations and aims of this thesis these are summarised by the following points:

1. To create a weak type checker which checks the grammatical categories of a specifiction, which could be used on formal or semi formal specifications.
2. To create a document rhetorical checker which checks a formal specification for loops in the reasoning and give warnings if there are outstanding preconditions to be totalised.
3. To automatically produce documents such as dependency and goto graphs to assist the users in analysing the system specification and to help with the translation into a theorem prover.
4. To create an easy step by step method to translate formal specifications into a theorem prover for userwho are novices in theorem proving. With each step in this path will be it's own correctness checker with some level of rigor.

2.1.1 To create a weak type checker which checks the grammatical categories of a specifiction.

The first point is outlined in chapter **??** and described in detail in chapter **??**. The weak type checker can check for grammatical correctness of formal and semi formal specification. A LaTeX package named `zmathlang.sty` has been implemented which allows the user to annotated their specification in weak typing categories. When the document is compiled the annotations then output coloured boxes around each of the categories in their colours which can be visually analysed by the user. An automatic weak type checker has been implemented to parse through the specification with it's annotation to check if the specification is correct or not. The automatic weak type checker follows a set of rules described in chapter **??**.

One limitation of the ZCGa checker is that the user needs to annotate their specifications by hand using the LaTeX package. This may sometimes be a repetitive and boring task and improvement to this limitation is described in section 2.2.2.1. Another restriction to this point is that although the ZCGa can weakly type semi formal specifications it can only check the parts which are written in a formal syntax. For example a *declaration* must be written in the form '`variable:type`' for the weak type checker to parse it. A more beneficial weak type checker would possible be able to parse over **informal** specifications. More on this idea is described in section 2.2.2.4.

2.1.2 To create a document rhetorical checker which checks a formal specification for loops in the reasoning.

The second achievement of this thesis was to create a document rhetorical checker which is described in detail in chapter **??**. The document rhetorical checker can check for any loops in the reasoning in the dependency and goto graph of the specification. The annotations for the ZDRa are implemented in `zmathlang.sty` which can be used on the specification to annotate chunks of the specification. When using this package to compile the document, boxes around each of the instances of the specification are shown and be analysed by the user. An automatic ZDRa program then parses through the annotations and checks the specification if it is ZDRa correct. Similarly to the ZCGa checker, the ZDRa checker is implemented in `Python`.

Like the ZCGa, the ZDRa annotations for the specification has to be done by the user. An more user friendly way to do this task would be a drag-and-drop idea where the user can highlight a piece of specification and click a button to add what instance this is. The relationships of the ZDRa could be done in a similar way. This could added to the current userface described in chapter **??**. A second limitation of the current ZDRa is that users can chunk any part of specification (formal or informal) the translation from the ZDRa annotated text can only be done from Z into Isabelle. It may be useful to translate from any formal specification into Isabelle (or any other theorem prover). More information on this extension is described in section 2.2.2.3.

2.1.3 To automatically produce documents such as dependency and goto graphs to assist users in analysing the system specification.

The third creation of this research is automatically produce documents which will be used to aid system engineers and software developers in analysing their system specifications.There is in total 5 items automatically produce in ZMathLang.

- dependency graph
- goto graph
- General Proof Skeleton aspect (Gpsa)
- isabelle skeleton
- halfbaked proof

The first 4 are automatically produced and stem from a ZDRa correct specification. The halfbaked proof can be automatically produced from a specification which is both ZCGa and ZDRa correct.

The dependency graph and goto graph (chapter ??) show how the instances are related to eachother, the Gpsa (chapter ??) show in which logical order the instances should be in order to be translated into a theorem prover with added instance to act as proof obligations. The Isabelle skeleton (chapter ??) uses the ZDRa instance names and creates a skeleton in Isabelle syntax. The halfbaked proof (chapter ??) is produced by using the Isabelle skeleton and the ZCGa annotated document. The filled in Isabelle skeleton is therefore the original specification translated in Isabelle syntax along with added proof obligations.

One limitation of the halfbaked proof is that not all mathematical Z syntax is translated into Isabelle using ZMathLang. The syntax which is translated is shown in table ?? in chapter ??. The current syntax covers all the examples which are in the appendix and in [3]. However the syntax for all of mathematics is large and more work can be done on translating more complpex mathematical syntax into Isabelle in ZMathLang. These can include schema hiding, piping, conditional expressions, Mu-expressions [12] etc.

The proof obligations created in the Gpsa are properties which check the consistance of the specification. These proof obligations are examples of properties which the user may wish to prove about the specification. Other complex proof obligations could also be added to ZMathLang, more details on this topic are described in section 2.2.2.2.

2.1.4 To create an easy step by step method to translate a specification into a theorem prover for novices in theorem proving.

The final accomplishment of this thesis is also the general aim of this thesis. The step by step method is outlined in chapter ??, which outlines how a user can get from a Z specification to a full proof in Isabelle. An example of this on a single specification is given in chapter ??. Each of these steps are decribed individually throughout this thesis. There are 6 steps to achieve a full proof for the specification in question. The first 2 steps require user input and automation, the last step requires user input and 3 steps in between are fully automated. By following this method it is easier to translate a specification into a theorem prover with no theorem prover knowledge up to step 5 (as described in chapter **??**).

However the limitation of this is that step 5 to step 6 requires user input and this stage requires some theorem prover knowledge. Proving lemma's in a theorem prover is not easy and requires expertise in the chosen theorem prover. Apart from the theorem provers own help tools (such as sledgehammer in Isabelle), future work may include investigating how to help users with this final stage. For example automating a way to show users which tactics they may find useful in proving a certain lemma. Another limitation of this outcome is that even though the user doesn't need Isabelle expertise to translate their specification into Isabelle they still need to learn the ZMathLang framework. This limitation can be aided with a user friendly interface and well documented guides such as [9].

2.2 ZMathLang Current and Future Developments

2.2.1 Other Current Developmets

The research on ZMathLang was started in 2013 and provides a novice approach to translating Formal specification to theorem provers. With this approach the gradual translation of the formal specification document is made via "aspects". Each aspect checks for a different type of correctness of the formal specification and output different products in order to analyse the system. Moreover, the annotation of the formal specification document should not require any expertise skills in the language of the targatted theorem prover. The only expertise needed for the annotations include the expertise of the formal specification document.

The ground basis of the MathLang framework were studied by Maarek, Retel, Laamar and various other master and undergraduate students under the supervision of F.Kamareddine and J.B. Wells. This thesis presents the ground basis of the ZMathLang framework which uses the methodology of the MathLang framework. The ZMathLang framework has taken the idea of breaking up the translation path from a document into a theorem prover and taking it through a grammar correctness checker, a rhetorical correctness checker, a skeleton into a proof. All the theory and implementation of the ZMathLang aspects have been developed and described in this thesis.

2.2.1.1 Other Developments

An extension to ZMathLang has started being developed by Fellar [7], [6] which takes the concept of ZMathLang and adds object orientatedness to it. With this, ZMathLang has the potential to translate not only Z specifications but object-Z specifications as well.

This thesis presents a very basic user interface to use with ZMathLang. Further developments on the user interface has been expanded during an internship by Mihaylova [9], [8]. The expansion on the user interface allows users to load and write their specifications. As well as going through each of the correctness checks, viewing the various graphs and skeletons all in one screen.

2.2.2 Future Developments

The future developments of ZMathLang have been discussed occasionally between students and supervisors during meetings. This section puts together and summarises these ideas and presents them to the reader in order to provide a general idea of future developments.

2.2.2.1 Automisation of the annotation

At present, the user needs to annotate their formal specification by hand using LaTeX commands before being check by the various correctness checkers. This sometimes can be a time-consuming task especially if the user isn't familar to LaTeX syntax. An advancement on this would be if the user would be able to visually see the Z specification as schema boxes (such as the compiled version of LaTeX) and then drag and highlight using mouse and buttons to annotate the specification with ZCGa colours and ZDRa instances. This idea could be done in a similar way to the annotations done in the original MathLang. Another way to ease the users input is if the labels would automatically label what user input. For example if the user labelled the variable '*v?*' as a term then all other variables '*v?*' would also be labelled a term automatically. This way the user wouldn't need to repeat the labels they have already done. This would drastically increse the workload for the user especially on very large specifications.

2.2.2.2 Extension to more complex proof obligations

The proof obligations described in this thesis are properties to check the consistancy of the specification. The current proof obligations for Z specifications are to give a flavour of what kind of properties to prove about the system and to ease the user in proving these properties. As mentioned before proof obligations for formal specifications is indeed a research subject in it's own right and more complex proof obligations can be developed to work alongside the ZMathLang framework. These proof obligation can come into the Gpsa part of the translation and follow through to the complete proof. If there are hint's or simple proof tactics to prove these properties then they can also be added to step 6 which would allow the user to get an idea of how to finish of the proofs.

2.2.2.3 Any formal specification to any theorem prover

This thesis describes how the ZMathLang framework can translate a Z specification into the theorem prover Isabelle. However, there are many other theorem provers which are prefered by certain users and ultimatly the ZMathLang framework should be able to translate from the Gpsa into a theorem prover of the users choice and not just be restricted to Isabelle. In this case steps 1 to 4 would be the same, regardless of which theorem prover the user wishes to translate to. The change would be made in step 5 when creating a skeleton of the specification in the chosen theorem prover. Other theorem provers which ZMathLang could transate to would be Mizar/HOL-Z/ProofPower-Z/Coq etc.

There are many other formal languages to write specifications in which could be another idea for future research. ZMathLang currently parses through Z specifcations however, further research could be done for ZMathLang to work on any formal language such as alloy, event B, UML or VDM. Investigation on whether the grammatical categories in the ZCGa or instances in the ZDRa would need adapting. Otherwise the current annotations would be suitable for any formal notation and only the implementation would need to be changed.

2.2.2.4 Informal specifications

A final future idea would be to combine parts of MathLang which handles mathematical documents written in part mathematics and part english and to translate informal specifications into theorem provers. With this idea, perhaps a TSa aspect would need to be adapted for informal specifications. So that a system specification written completely in english could be checked for ZCGa, ZDRa and ultimatly translated fully into a theorem prover.

2.3 Conclusion

This thesis presents an approach to translate a formal specification into a theorem prover in a step by step fashion. This new approach is aimed at novices at theorem proving which could learn by example on how to translate specifications. Proving the properties themselves is still a difficult task but a large chunk of the work is done already automatically by ZMathLang. By checking a system specification within a theorem prover adds a level of rigour to the planned system and therefore adds a degree of safety. Perhaps one day there will be a system which can parse through a specification written in natural language with diagrams and tell the user automatically if it is all correct and all conditions are satisfied. Perhaps one day, we will have systems with no bugs at all.

write conclusion chapter

Find year for atelier b proof obligation user manual

Bibliography

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

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

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

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

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

[6] 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.

[7] D. Feller. Using MathLang to check the correctness of specification in Object-Z. 2015.

[8] M. Mihaylova. ZMathLang User Interface Internship Report. 2015.

[9] M. Mihaylova. ZMathLang User Interface User Manual. 2015.

[10] G. Rossum. Python Reference Manual. Technical report, Amsterdam, The Netherlands, The Netherlands, 1995.

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

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

[13] University of Cambridge and Technische Universitat Munchen. Isabelle. `http://www.macs.hw.ac.uk/~lb89/zmathlang/`, May 2015.