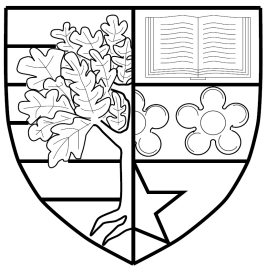
FROM FORMAL SPECIFICATION TO FULL PROOF: A STEPWISE METHOD

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

Proving formal specifications in order to find logical errors is often a difficult and labour-intensive task. This thesis introduces a new and stepwise toolkit to assist in the translation of formal specifications into theorem provers, using a number of simple steps based on the MathLang Framework. By following these steps, the translation path between a Z specification and a formal proof in Isabelle could be carried out even by one who is not proficient in theorem proving.

Acknowledgements

I dedicate this thesis to my loving and supportive boyfriend, Jeff.

Contents

List of Tables

List of Figures

Todo list

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

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 [54] which could be given to hardware or safety integrity system, as given by the International Electrotechnical Commission (IEC) [61] standard, is a SIL4. A SIL4 has a probability of failure of between 0.0001 and 0.00001 [21], 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 early 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 state space of a design and establish a correctness that is true for all possible inputs [17]. However due to the enormous complexity of systems these days they are rarely used as they may not be feasible.

Formal methods come in different shapes and sizes. The Abstract state machine (ASM) theory [10] is a state machine which operates on states or arbitrary data

structures. The B-method [19] is a formal method for the development of program code from a specification in the ASM notation. Z [71] is a specification languages used for describing computer-based 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.

Specification 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 [42].

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 into multiple smaller jumps, allowing the level 3 of rigour to be more accessible and thus more widely used.

1.1 Motivations

Mathematics is an ancient concept dating back to ancient Egypt and Babylonia times. Early mathematics was the study of measurements and quantity which influenced every day lives. As mathematics developed through logical reasoning, theoretical and applied mathematics it became more elabourate and complex. However no uniform notations ever became established.

In the late 19th century and beginning of 20th century, Russell & Whitehead [79] started to form a basis for mathematical notation. Their three volume work describes a set of rules from which all mathematical truths could be proven. In these early stages the authors try to dervive all maths from logic. This ambitious project was the first stepping stone in collaborating all mathematics under one notation.

Further to Russell & Whitehead's work, Bourbanki[1] wrote a series of books beginning n the 1935's with the aim of grounding mathematics. Their main works is

[1] A name given to a collective of mathematicians

included in the Elements of Mathematics series [11] which does not need any special of knowledge of mathematics. It describes mathematics from the very beginning and goes through core mathematical concepts such as set theory, algebra, function etc and gives complete proofs for these concepts.

With many advances in performing digitalised calculations the computer was born, in which enabled automatic proofs. With these new automatic proofs, there was an incredible need for consistent notations of mathematics. De Bruijn had this exact thought it mind when he worked on the AutoMath [26] project. AutoMath (automating mathematics) was the first attempt to digitize and formally prove mathematics which was assited by a computer. AutoMath is described as a language for formalising mathematical texts and for automating the validation of mathematics. The AutoMath project is what brought uniform notations and automated proof together.

Further work on de Bruijns formal language, Kamareddine and Nederpelt [40] described a way in which mathematical texts can be divided and annotated using *categories*. However, even with this work, it is still hard to develop proofs for mathematical and formal texts using AutoMath.

Therefore in order to facilitate the computerisation process (and assist in developing an automated proof), this thesis proposes smaller computerisation steps which allow the translation (and hence the correctness checking) of formal specifications into a theorem prover such as ProofPower-Z [6] or Isabelle [77].

The reason to break the translation path into simple steps is because the original path is difficult and requires high expertise in theorem proving and the translation from a formal language to a theorem prover.

1.2 Contributions

The focus of this thesis presents a tool support for ZMathLang. There are aspects of formalisations and proofs contributed in this thesis however they are smaller parts which come from the tool support built to translate specifcation into Isabelle. We summarise the contributions of this thesis in the following list:

1. Staged an approach to translating semi-formal and formal specifications into Isabelle with automatic assistance.

2. Built tools to enable this approach.

3. Formalised and proved properties of tools and method.

4. Evaluated method and tools on a convincing set of examples.

1.2.1 Staged an approach to translating semi-formal and formal specifications into Isabelle with automatic assistance.

Translating specifications into theorem provers has shown to be a great difficult task. Not everyone on the software project team will know how to computerise specifications. The main contribution this thesis presents a tool support to assist in the translating formal specifications into theorem provers. This new path has been designed for individuals who have no or little expertise in the chosen theorem prover. It is broken up so that each step in the path checks for some form of correctness of the specification. The checks become more and more rigorous the further one goes a long the path.

Our approach can also translate partially formal specifications into theorem provers. That is specifications written in natural language which is on it's way to becoming formalised. The line belows shows a semi-formal sentence taken from a specification describing a clock.

```
A clock tells the time:nat.
Time is on going therefore time' < time
```

Not only does our new approach assist with translating the text but it can also performs the various checks of rigor along the way. For example, the grammatical correctness of this document can be checked (see ZCGa chapter 4) without it needing to be fully formalised. The documents rhetorical correctness (see ZDRa

chapter 5) can also be checked without the need for the document to be fully formal. ZMathLang is one framework made up of many smaller tools to deliver it's main aim, which is to computerise a system specification.

1.2.2 Built tools to enable this approach.

Another contribution of this thesis is the tools which we have built in order to check for various degrees of correctness and to produce documents which could be used for the system in question.

Again the innovation in this research is that the tools can act upon system specifications which have been partially formalised. Some examples of the tools are listed below:

- A tool to check the grammar of the system specification.
- A tool which check the rhetorical layout of the specification.
- A tool which produces documents to order the chunks of the specification.

The first tool to check's the grammar of the document. To do this the user first annotates the specfication with grammatical labels and then uses the automatic grammar checker to check the labels. These labels are known as '*categories*' and describe the elements found in formal specifications such as '*terms*', '*sets*' and '*expressions*'.

The second tool checks the documents rhetorical correctness. That is, it checks for any loops in the reasoning. This tool can be used on a specification which is written formally, semi-formally or completely in natural language. The rhetorical checker chunks part of the text together and describes the relationships between each chunk. Similar to the previous tool, the user labels the text and then uses the program to check for any loops in the reasoning of the document. The chunks include sections which change the state of the specification, or output a message etc.

One of ZMathLang tools is able automatically produce documents which can be used to analyse the system. These documents are simple to understand by stake-

holders of the project and can be used to describe to clients, developers, managers etc.

1.2.3 Formalised and proved properties of tools and method.

A third contribution given in this thesis is a formal view on some of ZMathLang's tools (shown in chapter 8). The grammatical correctness checker for mathematics has already been formalised in previous research. Since the formal specification grammar checker is an adaptation of the mathematics one, we have presented a formal view on the adaptation. Rules and properties for the Document Rhetorical aspect have been implemented (see table 5.3 in chapter 5). We have identified these rules in order to reason with and show certain properties about them in chapter 8. Based on ideas from [82], we develop our own formalisation of the document rhetorical checking program as well as the products it automatically produces.

We give a formal view on the dependecy and goto graphs using various definitions to describe the relations between nodes and edges. We also give examples of the nodes and edges of an annotated text and describe its equivalent nodes and edges in the dependency and goto graphs. The formal definitions from [82] have been used to construct the adapted definitions and properties for the document rhetorical aspect and graphs presented in this thesis.

1.2.3.1 Automatically generated properties to check the sanity of the specifications.

Although not the main contribution of this thesis, we have designed and implemented a tool which automatically generates safety properties to prove in Isabelle syntax. The properties used in the examples are in the class of sanity checks in which the state invariants of the system specification are checked against all the state changing schemas.

There are many other properties one can check in systems such as properties across specifications, required properties of a single specification etc (see section 2.7 in chapter 2). However in this thesis we have supplied a formal definition for

the sanity checks of Z specifications and implemented it in our tool so that it is automatically added during the translation.

1.2.4 Evaluated method and tools on a convincing set of examples.

Another contribution this thesis presents is our approach shown on a convincing set of examples. We have used our approach on the following specifications

- BirthdayBook [71]
- Clubstate [22]
- ModuleReg [22]
- TelephoneBook
- VideoShop [22]
- A specification which fails ZCGa
- A specification which fails ZDRa
- Autopilot (A semi formal Specification) [16]
- The ZCGa type checker
- Vending Machine [6]
- Clubstate2 [22]
- ProjectAlloc [22]
- Timetable [22]
- Steamboiler [8]

We have analysed the complexity of each of these specifications and shown in detail a few case studies of translating these specification using the toolkit of ZMathLang. We have discussed how the complexity affects the translation and given details on lines of code in each specification, amount of different environments for each specification and the amount of annatations needed for each specification.

1.3 Outline

In chapter 2 we begin describing the origins of MathLang framework for mathematics (MathLang), its success and where it has been used so far. We then describe Z specifications and the tools available for it so far.

Chapter 4 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` [68], is thoroughly described and details of how the tool can be used are given.

Chapter 5 highlights another contribution of this thesis. An explanation of rhetorical correctness for a specification is given. Instances and how they relate to each other are described as well as how a user can annotate these facts into a Z specification. Examples are given for all relations and instances, and rules are provided 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 is also given.

Chapter 6 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.

Chapter 7 summerises the path 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 8 gives formal definitions of the ZDRa correctness checker, dependency graphs and GoTo graphs. We prove various properties about the ZDRa. 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 9 we give an overview on the user interface for ZMathLang and we explain how one can use ZMathLang on Z specifications. Explanations of how to use each aspect are given via examples and screen shots. The tables of output messages which a user can receive are highlighted and explained.

Chapter 10 goes through one entire specification (modulereg specification [22]) 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 11 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 doing the proof in one step compares against or multiple steps.

Chapter 12 evaluates the ZMathLang toolkit. Gives details on the complexity of the specifications we have translated and goes through some specification case studies.

Finally, a conclusion is presented in chapter 13 which summarises the contributions made in this thesis. The limitations of this research and potential areas of future research are also discussed.

Chapter 2

Background

2.1 MathLang for Mathematics

MathLang originally started in 2000. It's original goals was to allow gradual computerisation and formalisation of mathematical texts.

Maarek [56] describes MathLang in the 3 following points:

1. *MathLang is* **a framework.** It is meant to be used for communication as a concrete support for human mind formulation. MathLang is a well structured framework aimed to synthesize the common mathematical language.

2. *MathLang is* **for mathematics**. It is meant to be open to any branch of mathematics and to any topic that uses mathematics as a base language. MathLang mimics mathematics in its incremental construction of a body of knowledge.

3. *MathLang is* **for computerisation.** MathLang is meant to be a medium for a human-system, human-human via a digital support, and system-system communication. MathLang is a computer-based framework and therefore offers automation facilities.

MathLang is not a system for proof verification but a framework to computerise and translate information (such as mathematical text) into a form on which proof checkers can operate.

The MathLang framework provides extra features supporting more rigour to translation of the common mathematical language. One can define further levels of translations into more semantically and logically complete versions. This gradual computerisation method should be more accessible than direct formalisation, because a number of first levels do nor require any particular expertise in formalisation.

So far Mathlang has given alternative and complete paths which transform mathematical texts into new computerised and formalised versions. Dividing the formalisation of mathematical texts into a number of different stages was first proposed by N.G. de Bruijn to relate common mathematical language to his Mathematical Vernacular [26] and his proof checking system Automath.

2.1.1 The Current MathLang Design

The MathLang Framework instructs the computerisation process to be broken up into a number of levels called **aspects**. Each aspect can be worked out independently, simultaneously or sequentially without prior knowledge of another aspect. The current MathLang Framework contains three well-developed aspects, the Core Grammatical aspect (CGa), the Text and Symbol aspect (TSa) and the Document Rhetorical aspect (DRa), and has further aspects such as the Formal Proof Sketch.

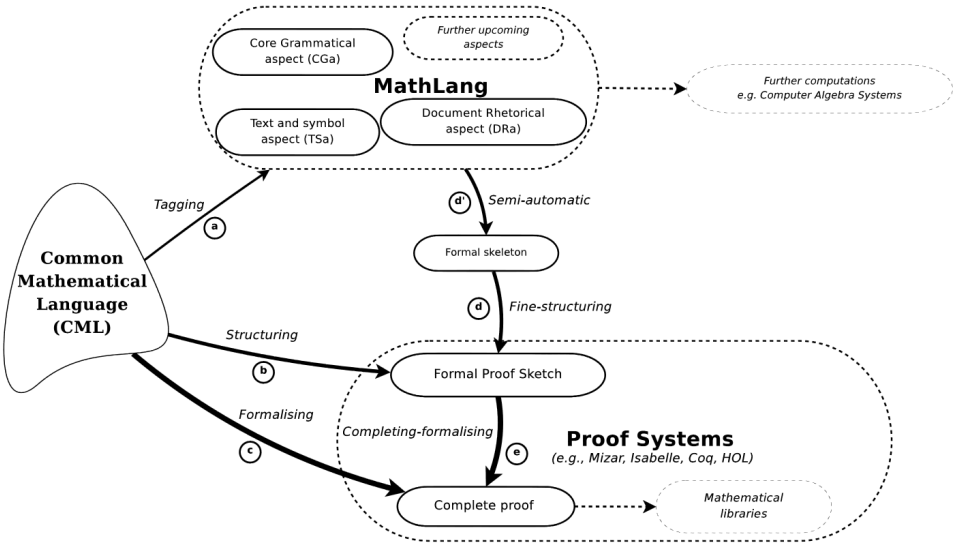


Figure 2.1: The MathLang approach to computerisation/formalisation [38]

Figure 2.1 shows the overall situation of work in the current MathLang Framework. The labelled arrows show the computerisation paths from the common mathematical language to any proof system. The width of the arrow representing each path segment increases according to the expertise required. The level of expertise needed to computerise a CML text straight into a complete proof is very high, however the level of expertise is much smaller by using the MathLang framework to help form a formal skeleton and then into a complete proof. The dashed arrows illustrate further computerisation that one can envision.

2.1.2 Core Grammatical aspect

The current CGa in MathLang uses a finite set of grammatical *categories* to identify the structure and common concepts used in mathematical texts. The aims of the CGa is to make explicit the grammatical role played by the elements of mathematical texts and to allow the validation of the grammatical and reasoning structure within the CGa encoding in a mathematical text. The CGa checks for grammatical correctness and finds errors like an identifier being used without and prior introduction or the wrong number of arguments being given to a function [66].

2.1.3 Text and Symbol aspect

The TSa builds the bridge between a mathematical text and its grammatical interpretation. The TSa is a way of rewriting parts of the text so they have the same meaning. For example some mathematicians may prefer to write "a=b and b=c and c=d", others may prefer "a=b, b=c, c=d" and some others may prefer "a=b=c=d". As you can see all these methods of writing have the same meaning however some symbols are different. The TSa annotates each expression in the text with a string of words or symbols which aim to act as the mathematical representation of which this expression is. This allows everything in the text to be uniform.

2.1.4 Document Rhetorical aspect

The Document Rhetorical aspects checks that the correctness of the reasoning in the mathematical document is correct and that there are no loops. The DRa mark-up system is simple and more concentrated on the narrative structure of the mathematical documents whereas other previous systems (such as DocBook [1], Text Encoding Initiative [2], OMDoc [3]) were more concentrated on the subtleties of the documents. It is used to describe and annotate chunks of texts according to their narrative role played within the document [66]. Using the DRa annotation system we can capture the role that a chunk of text imposes on the rest of the document or on another chunk of text. This leads to generating dependency graphs which play an important role on mathematical knowledge representation. With these graphs, the reader can easily find their own way while reading the original text without the need to understand all of its subtleties. Processing DRa annotations can flag problems such as cicular reasoning and poorly-supported theorems.

2.1.5 Full formalisation paths into Theorem Provers

The MathLang project started in 2000 when F.Kamareddine and J.B. Wells started the project within Heriot-Watt University as part of the ULTRA group [36]. It was an idea for a new mathematical language and framework to keep most of the advantages of Common Mathematical Language (CML) and avoid it's disadvantages. This framework would allow a gradual computerisation and formalisation of mathematical texts.

The framework was first set out in 2003 [55] and then saw an established path for conversion of a mathematical text written in CML into the isabelle proof assistant using rules and MathLang annotations [47]. A few short projects (by 4th year undergraduate dissertations or MSc students) have developed MathLang into the Framework it is today. A prototype of the Core Grammatical aspect and Text and Symbol aspect were defined in the PhD thesis of Manuel Maarek [56] and then a

[1] http://www.docbook.org
[2] http://www.tei-c.org/index.xml
[3] http://www.omdoc.org

gradual computerisation into the Mizar proof assistant using the three key aspects of MathLang were a great success and published [38].

The design of the CGa is due to Kamareddine, Maarek and Wells [39] and the implementation of the CGa is due to Maarek [56]. The design of the TSa is due to Kamareddine, Maarek, and Wells with contributions by Lamar to the souring rules [37], [56], and [48]. The implementation is primarily by Maarek [56] with contributions from Lamar [48]. The design and implementation of DRa were the subject of Retel's thesis [66]. Further additions have since been carried out by Zengler [41].

2.1.6 Conclusion

A lot of work has already been completed on the MathLang Framework. The three aspects, CGa, DRa, and TSa, have been designed and redesigned so that a variety of mathematical texts and symbols could be used. Then the aspects have been implemented for ease of access. A translation from a mathematical text into the Mizar proof checker has been worked through and described in detail in a published paper [38] and a PhD thesis [56]. A partial translation from a CML text into the Isabelle Syntax has also been carefully described in the 2009 paper [49] and also written in a PhD thesis [48]. Some of the future work described was to allow MathLang to be used as a tool to computerise other pieces of information, such as another piece of academic text yet it does not need to be mathematical.

2.2 Formal Methods in practice

Formal methods are mathematical approaches to software and system development which support the rigorous specification, design and verification of computer systems [30]. Specifications are a collection of statements describing how a proposed system should act and function. Formal specifications use notations with defined mathematical meanings to describe systems with precision and no ambiguity. The properties of these specifications can then be worked out with more confidence and can be described to the customers and other stakeholders. This can uncover bugs

in the stated requirements which may not have found in a natural language specification. With this, a more complete requirements validation can take place earlier in the development life-cycle and thus save costs and time of the overall project. The rigor using formal methods eliminates design errors earlier and results in substantially redecued time [32].

Abrial presents two case studies in [2] describing the use of formal methods in industry. He concludes that one of the problems is that some managers are afraid that engineers will not be able to perform the interactive proofs. This thesis proposes to address this problem by inventing a method for a thereom proving novice to translate a formal specification into the theorem prover with little or no knowledge of the chosen theorem prover. The research presented here proposes to provide an addition to testing and not a replacement. However the effort and costs should be reduced in the testing stages as the bugs would be found earlier in the specification and verification phase of the project. As well as giving the proposed system a highier level of rigor.

2.2.1 The use of Formal Methods in Industry

Despite these advantages some managers sometimes argue the cost of producing a system using formal methods do not cover the costs. However the rigour using formal methods eliminates design errors earlier and results in substantially reduced time. Investing more effort in specifying, verifying and testing will benefit software projects by reducing maintenance costs, higher software reliability and more user-responsive software [18].

Even in the 21st century we still experience a 'software crisis' where software projects are being delivered far behind schedule, quality is poor and maintenance is expensive. This software crisis allows for bad software to be released e.g. the computer system 'Sabre' [65], which went off-line for almost a day leading to the cancellation of more then 700 flights.

Well engineered software is software which is suitable, efficient, reliable and maintainable with low costs and on schedule.

The cost of testing is around 50% of the entire project cost. Maintenance cost is 2-4 times greater than pre-delivery cost. In large projects, failure to find and correct software errors can increase the cost of the software by 100 times, in small software projects it's usually about 2-4 times more [35]. Therefore more effort should be spent in requirements analysis and design to catch errors early in the project life-cycle. The computerisation of formal specifications using the MathLang framework should help with requirements analysis, as it is concentrating more on getting the requirement specifications correct and thus minimising errors later on in the project life cycle. For example, in the Sholis project [43], using a formal specification was most effective for fault finding, therefore if the specifications are correct, then the program implemented should then in turn contain less errors if it follows the correct specification.

King, Hammond, Chapman and Pryor's paper [43] was based on the SHOLIS defence system. It highlighted the importance of having a formal specification on a system to check for errors. It was found that the Z proof was the most cost effective for fault finding. The Z specification found 75% of the total faults for the system. Since Z specifications are important for finding faults in SIL4 systems (based on the sholis project), then checking the correctness of the Z specification is itself very important. Note that the specifications found 75% of errors and not 100%. As human error can still occur in formal specifications, using the ZMathLang approach may increase the percentage of errors found.

Hehner [34] also supports the use of specifications. He states it is the job of the specification to distinguish those things that are desirable in the program however when looking through a specification just with the human eye [43] it is easy to make errors. Which is why checking the correctness of a specification through a theorem prover would help.

One reason industry may be reluctant to use formal methods is because they might perceive that the methods are too complicated for the benefits gained. Simplifying these formal methods so that anyone could understand may pose a great benefit and thus may be used more often in industry.

Hehner questions if all programs should have specifications. Which leads to the question of should simple programs also have specifications as well as high integrity systems. The benefits of planning and specifying a program far outweigh the time and cost of catching bugs and errors at a later stage [53] of complex programs. However it may be too time consuming for smaller program and it would be up to industry leaders themselves to decide.

Specifications, Programs, and Total Correctness [34] outlines that a programming language should not be the specification language. As not all industry experts are programming experts, the specification should be open for everyone in the team to understand the program not just the developers.

Hehner also states that total correctness is a mistake and partial correctness is enough. This may be because some software such as on aircraft's need to be running all the time when in the air and do not need to terminate. However with other programs it is important that they do terminate as a safety feature for example the automatic track gauge changeover for trains [4]. It is important that the program should terminate if anything should happen such as errors or a crash, therefore total correctness of only some programs is needed.

An Introduction to Proving the Correctness of Programs [33] describes the specification of correct behaviour of a program by the use of input/output assertions. This would be very expensive in very large programs. So it may be good for smaller programs only. Assertion is a very basic approach.

However with this approach, checking for correctness can only be done by an expert in that particular programming language. They will have to understand where a procedure starts and where a procedure ends. By checking the correctness of specifications using the ZMathLang method it will allow for many program specifications to be checked by a wider audience and not just expert programmers.

The assumptions used in the example on page 336 [33] resulted from an unresolved execution of the IF statement.

A paper reflecting on industry experience with proving properties in SPARK **??**, describes a programming language and verification system that will offer sound

verification for programs. It states that SPARK and the use of proof tools remain a challange (published in 2014) as the 'adoption hurdle' is percieved too high. Customers and regulators have taken a variety of stances on static analys and theorem provers. Where some places in industry have adopted the idea others remain sceptical. Hopefully this thesis will present an idea on how formal analysys could be simplified and broken up into smaller more understandable steps and thus would allow more users to take on the idea.

2.3 Methods for checking for Software Correctness

Traditionally, functional correctness has been obtained with pen and paper or an interactive proof assistant. Well-designed program verifiers are reducing the effort involved in doing full verifications.

Proof assistants sometimes limit which program properties they reason about. By using the ZMathLang framework the specification would undergo different levels of rigor (and thus different types of checks) for example one might only want to check the grammatical correctness of the specification or one might want to fully formally prove the specification, different projects require different amounts of verification therefore the ZMathLang will allow this choice.

Dafny [52] features modular verification, so that the separate verification of each part of the program implies the correctness of the whole program. This is similar to ZMathLang, where Dafny checks different parts of the text and thus confirms correctness of the full text, ZMathLang checks the correctness of the text through different levels of rigor to imply and fully correct specification.

Dafny was able to do a proof for the Schorr-Waited Algorithm, however the writer states that the loop invariants are complicated because they are concrete. A refinement approach such as Jean-Raymond Abrials [1] may be preferred in this case.

Another attempt at checking for correctness was written by Rex L.Page in Engi-

neering Software Correctness [63]. A general theme within this paper, is that design and quality are important in engineering education. When teaching students how to create programs, it is not enough just to teach them how to develop software but to how develop good quality software. This paper describes experiments with the use of ACL2 (a subset of lisp), which is embedded on mechanical logic to focus on design and correctness. Using ACL2 emphasises the importance of software design and correctness.

ALC2 is coded therefore users must know how to code software to formulate proofs. The intention of ZMathLang is to allow many people in the development team to be able to formulate proofs such as project managers, designers, engineers etc. Therefore no coding is neccessary and no new programming language is needed.

PVS (Prototype Verification System) [62] is an environment for constructing specifications and developing proofs which have been mechanically verified. PVS has it's own specification language, which engineers would need to learn as well as using the environment for proofs. Type checking is undecidable for the PVS type system. The PVS also provides a language for defining high-level inference strategies.

Another tool which analysis the Z notation is Hol-Z [13], which is also a proof environment for Z. Hol-Z is embedded in Isabelle/HOL therefore it provides a Z type checker, documentation facilities and refinement proofs with a theorem prover. The Z specification is implemented in LaTeX then typed checked using an external plug in Zeta, it is then transformed into SML files to be added into the Hol-Z theorem prover environment. The user will need to have some good expertise in using the Hol-Z proof environment in order to fully prove the specification.

Fuzz [74] is another example of a typechecker for the Z language. It includes style files for LaTeX and checks for the logical correctness and Z type correctness of Z specifications. This is different to the ZCGa type checker as the weak types in ZCGa check for the grammatical correctness and not the full logical correctness of Z. Therefore the grammatical correctness of partially formal specification can aslo be checked. The ZMathLang framework presented in this thesis uses the 'zed' LaTeX

style package to typeset the Z specifications in the documents.

There are many other tools for Z which can be found on the Z Notation Wikia page [12]. For this thesis we will concentrate on translating Z specifications into theorem prover Isabelle [77]. Isabelle is a generic proof assistant which allows mathematical formulas to be expressed in a formal language and includes numerous contributions worldwide. It contains a large mathematical toolkit (majority of Z can be represented in Isabelle) and has a lot of support in forms of documentation and online. It is distributed for free, easy to find, download and install and is regularly updated. The original MathLang has translated mathematical texts into Isabelle already ([49]) therefore we can use parts of that research to aid the research in this thesis.

2.4 Proof carrying-code

Proof carrying code (PCC) is a framework for the mechanical verification of safety properties in machine language programs. The provider of the PCC must provide both the executable code and a machine-checkable proof. This is to ensure the safety of the executable code so that it doesn't access any other data it is supposed to. Appel [5] attempts to make PCC easier by using foundational method where he avoids any commitment to a particular type system or a verification checker. ZMathLang uses a similar approach as the specification goes through different types of correctness checks and only at the very end, picks a verification checker to translate into. All the steps until the final three (see figure 3.1) do not require the user to commit to a particular theorem prover.

PCC has several characteristics that allow it to execute foreign code safely. A critical components of any PCC implementation is the safety policy which is specified at the start, before any implementation takes place. This policy uses safety rules that the consumer of the machine code desires for any untrusted code. PCC is a two stage verification process [60], using a 'proof producer' and a 'code consumer' to do the work. The proof producer must produce two kinds of work, one is the machine code and the other is the proof to verify that the machine code is safely

executable. PCC is slightly different to ZMathLang as PCC includes the proof with the code of the program. Since a large system may have lots of different components which join to make one large system. The proof carrying code would need to be implemented in all of the components (or the most safety-dependent) components. Formal specifications can display and abstract view of the entire system and it's individual components. Therefore the entire system as a whole could be checked. However added rigor, one can use the ZMathLang framework to check the correctness as a whole system and then using proof carrying code to check the code itself for correctness.

Manish Mahajan [57] explains that any implementation of PCC must contain at least 4 elements: (1) formal specification language used to express the safety policy, (2) a formal semantics of the language used by the untrusted code, (3) a language used to express the proofs and (4) an algorithm for validating proofs. ZMathLang's DRa and CGa could be useful in creating and checking the formal specification needed to express the safety policy, (1), which again could add a level of rigor to the system. Mahajan also writes that the size of binary will be increased due to inclusion of the safety proof, this safety proof can be done during the formal specification part of the project, thus minimizing the size of the safety proof needed for the PCC.

2.5 Z Syntax and Semantics

The Z notation is based on set theory and mathematical logic. It is a particular formal method which was developed to specify the new Customer Information Control System (CICS) functionality [76]. The set theory includes standard set operators, set comprehensions, cartesian products and power sets. Z also has other aspects such as schemas which are used to group mathematical objects and their properties. The schema language can be used to describe the state of a system and ways in which that state may change [81].

In the Z notation there are two languages: the mathmatical language and the schema language. The mathematical language is used to describe various aspects of a design: object, and the relationships between them. The schema language is

used to structure and compose descriptions: collecting pieces of information and naming them for re-use. A schema consists of two parts: a *declaration part* and a *predicate part*. The *declaration part* consists of declared variables and the *predicate part* describes the variable values. We can write a schema either horizontally (figure 2.2) or vertically (figure 2.3).

$$Schema\ Name \mathrel{\widehat=} [declarations \mid predicates]$$

Figure 2.2: An example of a schema written horizontally.

Schema Name

declarations

---

predicates

Figure 2.3: An example of a schema written vertically.

If we wanted a property of some system which consists of two variables x and y and state that x must be smaller than y then we can write:

xLessThany

$x : \mathbb{N}$

$y : \mathbb{N}$

$x < y$

The full language of Z can be explored in [71], [22] and [81].

The semantics of Z has helped in the design of better specification languages by allowing critical comparisons of specification techniques. The semantics of Z [73] gives us a head start in formulating the different aspects of ZMathLang. The study in the semantics of Z provide a foundation for reasoning about specifications.

In [73] it states that many of the proofs needed during the development process are '*very shallow but contain a mass of detail*'. This detail may be difficult to understand by some stakeholders or team members in the system design team. ZMathLang would aid in this as it is a tool which produces documents which anyone

in the development team/clients would be able to understand.

The paper also states that *consistency, completeness and refinement are essential to program development.* Therefore if the specification is consistent, complete and refined then the program will be as well (as long as the program sticks to the specification). Which makes it very important to have the specification checked as well as the program for errors. Spivey also says that '*theory of signatures is decidable and therefore well suited to mechanical checking using a proof checker such as Mizar or Coq*'. However Mizar and Coq usually requires a lot of expert level knowledge to prove the specification directly. Therefore a major aim of this thesis is to develop a path to break up the translation into a theorem prover such as Isabelle, Coq or Mizar.

This thesis uses a lot of the specifications in Ed Currie's book, An Essence of Z [22]. The book would be a beneficial text to check for correctness as it is used in academia to teach students Z. The framework developed in this thesis is for novices to get to grips with checking the correctness of Z specification and translating them into a theorem prover. It does not promise to turn novices into theorem prover experts overnight but give them a guide with verifying the correctness of specifications. The research presented here is also not intended for theorem prover experts however users who are experts in theorem proving may also find some steps of the ZMathLang framework beneficial such as the ZCGa or ZDRa. The Essence of Z is also starts with very basics of logic to larger specifications which can be used a real software systems. It contains more then one specification and therefore gives a variety of syntax to use the ZMathLang framework on.

2.6 Types and their desirable properties

Type systems are good for many things [64], one of which includes that it allows early detection of some programming errors. Errors that are detected early can be fixed straight away rather then it lingering around to be discovered much later. Errors can often be pinpointed more accurately during type checking more often then in run-time, when their effects may not become visible until after some things go wrong,

which in high integrity systems can have disastrous results. Type systems support the programming process by enforcing disciplined programming. Type system form the backbone of the module languages used to packaged and tie together all the different parts of large scale software systems. Types are also useful when reading programs. They form a useful documentation to the reader about the behaviour of the program, this form of documentation can not be outdated like comments since when the program specification changes so does the types involved.

Type-free lambda calculus [7] allows for every expression to be applied to every other expression, eg $\mathrm{I} = \lambda\, x.x$ may be applied to any argument x to give the same result x. The expression may also be applied to itself. There are also typed versions of lambda calculus introduced by Curry [23]. Types are usually objects of a syntactic nature and may be assigned to lambda terms. Using Types in this nature, this thesis describes a way in the ZCGa (chapter 4) assigns grammatical types to parts of a specification written in Z or partially written in Z. By doing so, the grammatical correctness of a system specification could be checked. The grammatical types are an adaptation from the grammatical categories taken from [40]. One of the main benefits of the ZCGa is it can check the grammatical correctness of partial formal specifications, that is specifications which are written in natural language and are on the way to becoming formal. Other Z type checkers such as Z/Eves [69] and Fuzz [74] check the logical type correctness of a fully formalised Z specification.

2.7 Generating properties to prove for Formal Specifications

Definition 2.7.1. *A logical formula associated to a correctness claim for a given verification property. The formula is valid if and only if the property holds. The correctness of the property under verification is delegated to proving the correctness of the new formula [51].*

Therefore a proof obligation is a logical formular which the specifier must show to be a consequence of the specification so that a specification can be taken to be

acceptable. In a more pragmatic sense proof obligations may be viewed as what the developer of a specification is obliged to prove in order to confirm that development is consistant.

Woodcock and Cavalcanti [80] use the alphabetised relational calculus to give denotational sematics to different constructs from programming patterns. This paper describes '*healthiness conditions*' which identifies properties that characterise theries. Each one of these healthiness conditions represents a fact about the computational model for the programs being studied.

Example 2.7.1. *The variable '*clock*' is an observation of the current time which is always moving onwards. The following predicate 'B' specifies the clock variable:*

$$B \mathrel{\widehat{=}} clock \leq clock'$$

The healthiness condition described in this paper are specific proof obligations for the concept of Unifying theories of programming (UTP). The semantic model of UTP is presented a Z specification. Therefore the healthiness conditions for the specification would in one way be checking for the correctness of the UTP model. In a similar way, it is important to add healthiness conditions or as we call them in this thesis, 'safety properties' in order to check each individual specification for various types of correctness.

Stepney describes two proof project written in Z in her paper a tale of two proofs [75]. She explains how the proof process is deelply affected by **why** something is being proved, **what** is being proved and **how** the finished proof is to be presented. Stepney suggests that the proofs for specifications themselves do not have to be deep but the workings of what to proves can add to the labour. It is also important to keep in mind how deep the customer wants the proof and what level of assurance they need. She highlights 5 different points to prove:

1. **Consistency checks**: Prove that your specification is cosistent and that it has a model.

2. **Sanity Checks**: In a 'state and operations' style specification, prove that the state invariants are satisfied throughout and that the precondition of each operation is not *false*.

3. **Emergent properties of a single specification**: Make explicit as a theorem some desired or suspected property of the specification, then prove it holds.

4. **Required properties of a single specification**: If some property is required to hold of the specification, and the specification has captured it implicitly, it needs to be made explicit and shown to hold.

5. **Properties across specifications**: Prove that a certain relationship holds between two specification such as the refinement relationship.

Figure 2.4: Different points to prove in a specification [75].

Some of the points in figure 2.4 would be fairly easy to automate such as 'consistancy checks' and 'santity checks' however the other 3 points would be difficult to automate as they would all depend on the specification in question and would perhaps need some *extra information* to decipher these properties. For example if one wish to automate proof obligations from point 5 the user would need to implement another specification (such as a refinement specification) and then prove the relationship between the original specification and the new one. In this thesis, we will concentrate on properties described in point 1 which have been automated.

Stepney also explains that if the development process is incremental, it would be worthwhile getting a good structure for a proof up front which you can add more details in later. ZMathLang can automatically generate this first structure which could be added to if needed. It can produce a specification in Isabelle syntax where other properties and details could be added to by the user at a later stage.

Most recently Mark Adams [3] describes that even formalisation itself can be

prone to error and therefore even if we do get a fully proven specification, the proof will also need to be checked. He outlines the flyspeck project which assists with the checking of formalised proofs. Adam calls this '*proof auditing*', which adds another step of rigour to the theorem. ZMathLang assists with translating the specification into a theorem prover along with consistancy lemmas to prove. The user can then choose to prove these lemmas and the proof audit their thereom. However, even if proof auditting adds another level of rigour it is important to keep in mind who the user is doing the proofs for. As Stepney pointed out in [75], some clients wouldn't need that amount of rigor for their projects and only the proved safety properties may be enough.

Woodcock et al also describes that a specification can be developed in such a way that can lead to a suitable implementation called refinement [81].

To refine a formal specification, more data must be added e.g. how certin calculations should be carried out. He states an abstract specification may leave design choices unresolved and its up to the refinement specificatino to resolve some of these choices. An example of this is shown in the following:

Example 2.7.2.

ResourceManager

$$free : \mathbb{F}\mathbb{N}$$

Any resource that is free can be allocated

Allocate

$$\Delta ResourceManager$$
$$r! : \mathbb{N}$$

$$r! \in free \land free' = free \setminus \{r!\}$$

If there is more than one resource free in example 2.7.2 then this will class the specification as non deterministic.

Example 2.7.2 can be refined in that if there is more than one resource free, the resource with the lowest number should be allocated. This is shown in the following example:

Example 2.7.3.

Allocate

$\Delta ResourceManager$

$r! : \mathbb{N}$

$r! \in min\, free \land free' = free \setminus \{r!\}$

Refining an abstract specification which is descibed in [81] and in [71] is exactly what Stepney points out in point 5 from figure 2.4. To show that this property holds the user would need to produce another specification which refines their original one and then include propertie of how they relate to eachoether. Since each individual specification is different then refinement specifications would be different to. Thus we wouldn't be able to automate this point. ZMathLang aims to assist the user in translating and proving their specification, the proof effort will be focused on properties which can be automated e.g. item 1 in figure 2.4. Items 3, 4 and 5 would be to difficult to automate and therefore out of the scope of this thesis.

Fraser and Banach [31] state that model based formal methods usually come in the form of many incompatible tools. Therefore they devised a system to combine different techniques called the frog toolkit.

Many verification tools today tend to utilize a single technique and are unable to ineract easily with other tools in other kits. A more dynamic approach such as RODIN [20] and Overture [50] are persuing a more flexible approach. ZMathLang will need to follow in the footstep of these two toolkits in order to be succesfull and not commit to a particular system until needed. This is why the steps 1-3 of the translation path (figure 3.1) are generic and can be used to translate the specification in any theorem prover. It is only at step 4 which is where we start committing to

Isabelle.

The main aim of the frog toolkit was to support retrenchment, which is a formal technique used along side refinement. The new toolkit was created which was able to use a variety of techniques in a single working environment. The frog toolkit can prove the relationshipis between multiple specifications (which we see as point 5 in figure 2.4 from Stepney [75]). To do this the user should have at least 2 specifications implemented to prove the relations (usually one is a refinement specification of the first one). These specifications where then written in frog-ccc, a meta language to use within the frog toolkit. ZMathLang's aim is to prove the properties described as point 1 in figure 2.4 therefore another specification to refine the abstract one is not needed.

Since formal specifications are not executable it is difficult to verify the consistency of the specification. Wen, Miao and Zeng [78] present and approach to generate proof obligations to check for consistancy of object Z specifications. Checking for consistancy of specifications is described in point 2 in figure 2.4. In their paper, the authors explain two types of proof obligations which check that a object Z specification is non conflictive. They are:

- Existence of initial state
- Feasibility of an operation

The initial state is a state within the state space of the specification which should exist and satisfy the stateInvariants. In all specifications their should exist a state which initialised the beginning state of the specification.

An operation can tranform one state to another. Therefore if the operation is feasible, the pre state (state before the state change) and post state (state after the state change) should always satisfy the state invariant.

We will use the definitions shown in [78] to automatically generate the proof obligations to check for the correctness of our Z specifications as the specifications we use in this thesis are all state based.

In summary there are many different approaches to finding properties to prove about formal specifications. Some which can be easily automated, some which need

some more information and some which will need to be written manually. Since we do not expect the users of ZMathLang to be experts in the fields of theorem proving we want to attempt to automate proof obligations which can be easily understood by the user. Then once the user gains some knowledge of their theorem prover, they can add other proof obligations manually. As explained in this section, it is important to note who the user is doing the proofs for, as the customer may not need or want complex proofs. Therefore we shall generate the proof obligations which can be automated and do not require any extra information to be automated.

2.8 Conclusion

This chapter has described in depth the reasearch old and new which has relevence to this thesis. The MathLang framework which was designed and implemented for mathematics has been described and gives details of the method in which plain mathematical documents are formalised and translated into theorem provers. In this section we show a diagram which was proposed to identlify the steps in which a complete proof can be achieved from a mathematical text.

The next section of this chapter highlights formal methods and formal languages used in practice. Formal methods which have been used in an industrial setting has been explained and analysed. We then outlined other methods for checking for software correctness which may or may not have been used in practice but also investigated in an academic setting.

Since this thesis concentrates on Z specifications we have outlined what are Z specifications and compared it with other formal languages.

We also researched into types and their properties since the first aspect of ZMath-Lang is heavily based on a weak typing system.

This chapter concluded with different ways one can generate properties to prove their formal specifications.

Chapter 3

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.

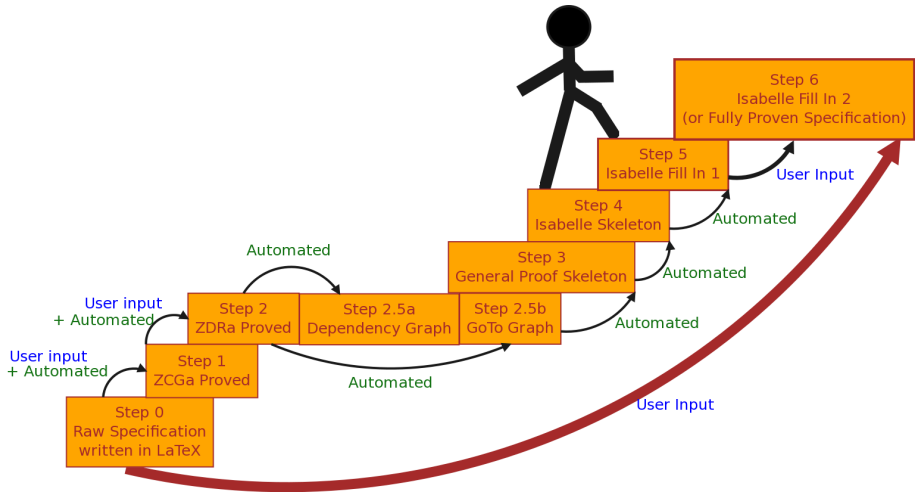


Figure 3.1: The steps required to obtain a full proof from a raw specification.

The framework is targeted at beginners in theorem proving. The users should have some idea of formal specifications but no or little knowledge of the targetted theorem prover. Figure 3.1 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. 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 has little or no expertise in checking for correctness by a theorem prover.This tool could also aid user 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 small black arrows represent the amount of expertise needed for each step. The last step the arrow is slightly thicker as some theorem prover knowledge is needed. However these arrows are still small in comparison to the red thick arrow which represents the translation in one big step.

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
- 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

3.1 How far does the automation go?

Figure 3.2 shows a diagram showing how far one can automate a specification on either side. ZMathLang is a toolset which assists the user 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 specification (going from right to left) in the diagram.

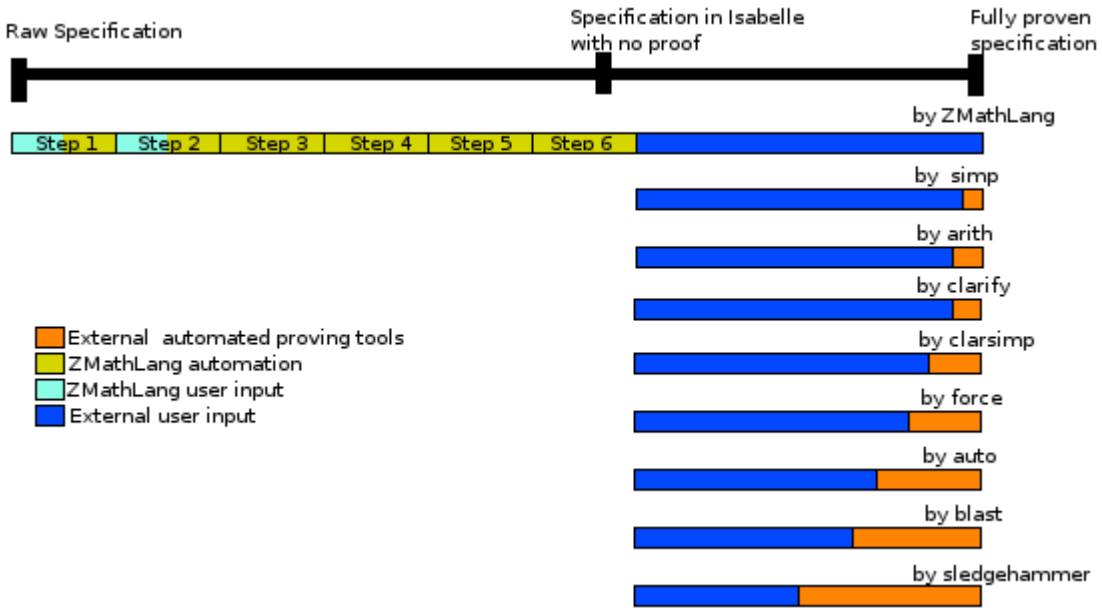


Figure 3.2: How far can one automate a specification proof.

In figure 3.2 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 (ZCGa and ZDRa) however the rest 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 yet 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 chapter 6 section 6.3). 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 3.2 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.

Even external automated theorem provers have their limitations. For example, the user can use the Isabelle tool '*sledgehammer*' assists the user automatically solving some proofs, but not all. The sledgehammer documentation advises to call '*auto*' or '*safe*' followed by '*simp_all*' before invoking sledgehammer. Depending on the complexity of your proof, this sometimes may prove the users properties, other times it may not and the user will still need to invoke sledgehammer to assist in reaching 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 further by automating proofs, 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.

3.2 Overview of ZMathLang step by step

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

3.2.1 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 the way to becoming formalised in Z). The specification should be written in LaTeX format and can be a mix of natural language and Z. An example of a specification written in the Z notation can be seen in figure 3.3.

$[NAME, DATE]$

BirthdayBook

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

$birthday : NAME \pfun DATE$

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

InitBirthdayBook

$BirthdayBook'$

$known' = \{\}$

AddBirthday

$\Delta BirthdayBook$

$name? : NAME$

$date? : DATE$

Figure 3.3: Example of a partial Z specification.

3.2.2 Step 1- The Core Grammatical aspect for Z

The next step in figure 3.1 shows 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 categories and then checks these for correctness. Figure 3.1 show 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 difference colours and can help identify grammar types to other members interested in the specification. A ZCGa annotated specification is shown in figure 3.4. The ZCGa is further explained in chapter 4.

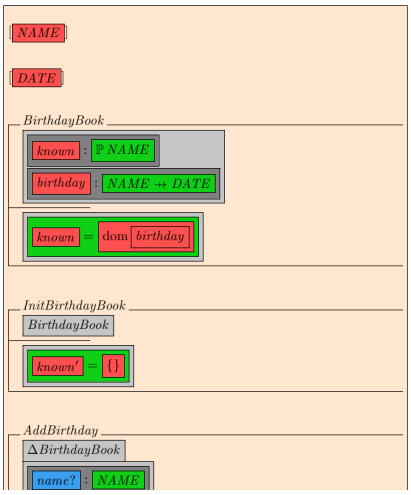


Figure 3.4: Example of a ZCGa annotated specification.

3.2.3 Step 2- The document Rhetorical aspect for Z

The ZDRa (chapter 5) step shown as step 2 in figure 3.1 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 and allows the user to describe the relationship between these chunks of specification. 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 give 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 3.5.

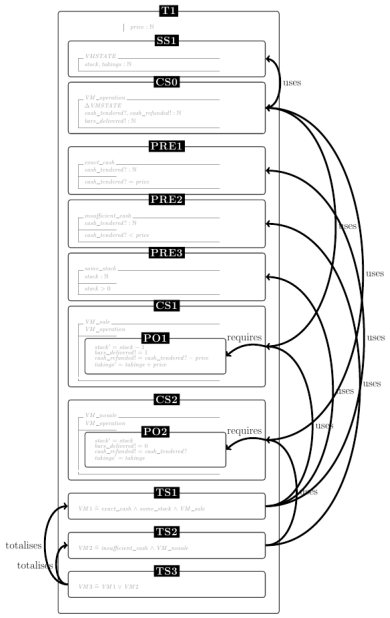


Figure 3.5: Example of a ZDRa annotated specification.

The ZDRa automatically produces a dependency and a goto graph (section 5.2.3), these a shown as 2.5a and 2.5b respectively in figure 3.1. The loops in reasoning are checked in both the dependecy graph and goto graph. An example of a goto graph is shown in figure 3.6.

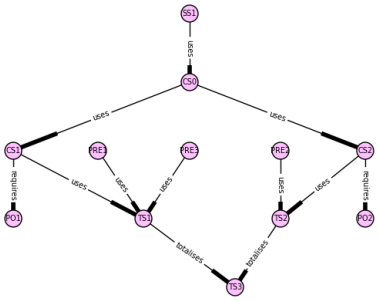


Figure 3.6: Example of an automatically generated goto graph.

3.2.4 Step 3- The General Proof skeleton

The following step is an automatically generated 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 not conflictive each part. An example of a general proof skeleton is shown in figure 3.7. The Gpsa is further described in section 6.1.

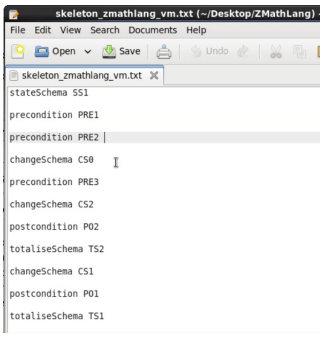
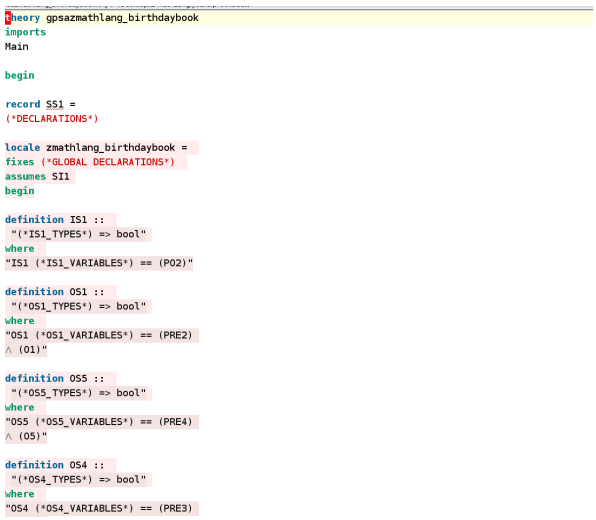


Figure 3.7: Example of a general proof skeleton.

3.2.5 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 3.8. Details of how this translation is conducted is described in section **??** 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 3.8: Example of an Isabelle skeleton.

3.2.6 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 proofs. An example of a filled in isabelle skeleton is shown in figure 3.9.

```
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 3.9: Example of an Isabelle skeleton automatically filled in.

In this case the Isabelle skeleton will not change. Further information on the translation is described in section 7.3 of this thesis.

3.2.7 Step 6- A fully proven Z specification

The final step in the ZMathLang framework and the top of the stairs from figure 3.1 is to fill in the Isabelle file from step 5. This final step is represented by a slightly thicker arrow in figure 3.1 compared with the others as the user may need to have some little theorem prover knowledge to prove properties about the specification. 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 in Isabelle as the user would allready have examples of other instances in their Isabelle syntax form. More details on this last step is described in section 7.4 of this thesis.

3.3 Procedures and products within ZMathLang

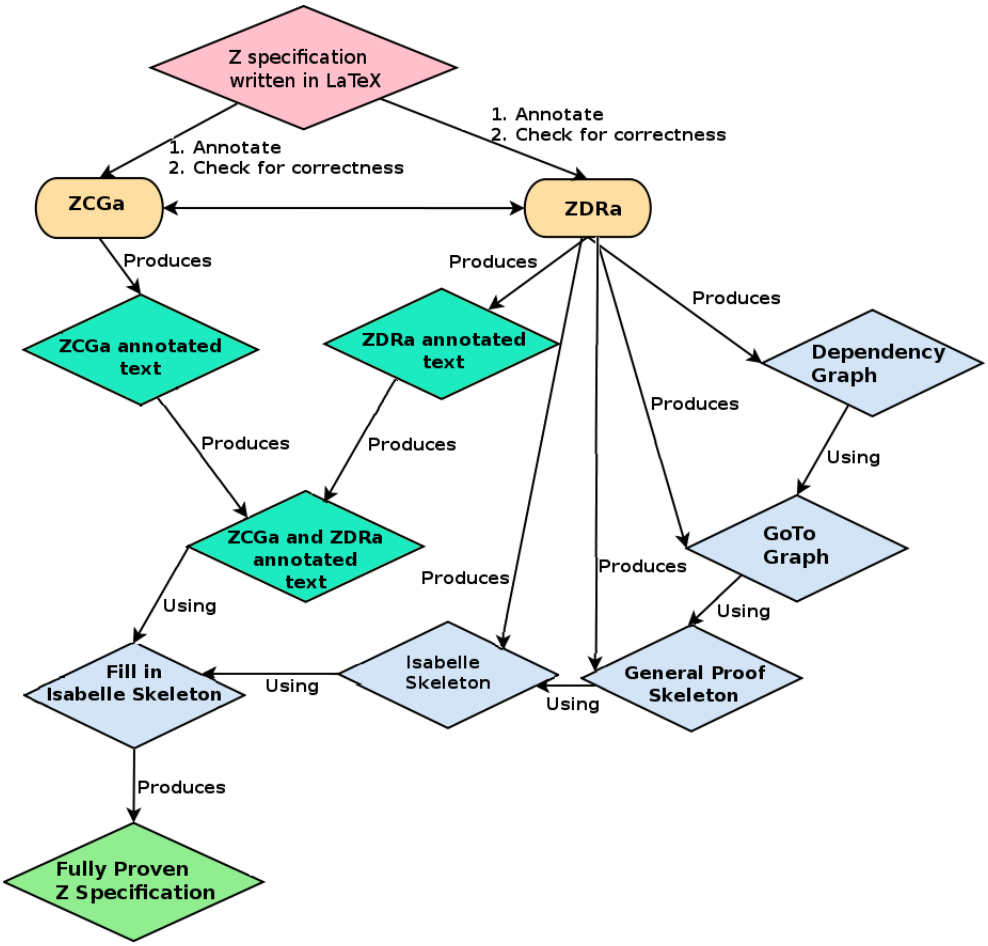


Figure 3.10: Flow chart of ZMathLang.

Figure 3.10 shows a flow chart describing the documents produced from using the framework and which parts are fully automated, partially automated and user input. 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 and produces a 'ZCGa annotated text'. The ZDRa procedure requires user input to annotated and the check is automated. 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 to obtain a 'fully proven Z specification'.

3.4 The ZMathLang LaTeX Package

The ZMathLang LaTeX package (shown in appendix B) was implemented to allow the user to annotate 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`.

3.4.1 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 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 3.11: 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 3.11). Where `*NameOfZCGaType*` is the name of the category i.e. 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*.

3.4.2 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. 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 3.12 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 3.13 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 3.12: Incorrect annotating of ZDRa.

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

Figure 3.13: Correct annotating of ZDRa.

It is important to note this embedding order as by annotating a chunk of specification using the ZDRa annotation `\draline` it keeps the inside of the part inside as maths mode. Since the annotation `\draschema` is outside the zed commands (eg `\begin{schema}`) then the zed commands make everything inside the instance into math mode.

```
\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 3.14: 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 3.14. 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.

3.4.3 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 5, however this section of the thesis describes the LaTeX commands implemented so that they can be used to annotated a specification in ZDRa.

```
\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 3.15: The syntax to define a ZDRa schema relation in the ZMathLang LaTeX file.

Figure 3.15 shows how the command `uses` has been implemented. The command takes 2 arguments (which are 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.

3.4.4 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 3.16: The syntax to define a ZCGa grammatical categories.

Figure 3.16 shows the commands to define the coloured boxes for *declaration* and *set*. As set is already defined in the mathematical LaTeX library, we redefine 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 as it is already in mathmode.

3.5 Conclusion

In total there are 6 steps in order to translate a Z specification into the theorem prover Isabelle. Each of these steps assist the user in understand the specification more, 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. The next chapter begins to describe step 1 (ZCGa) in more detail.

Chapter 4

Z Core Grammatical aspect

The ZCGa is a weak type checker, which checks for grammatical correctness in fully formal Z specifications and partially formalised Z specification. It is not the same as pure Z type checkers as it only checks the grammar on a sentence level and not the logical correctness. The ZCGa has it's roots in weak type theory for mathematics [40] and has changed for Z specifications. Core grammatical correctness for Z has also adapted the rules from the CGa for mathematics (see section 2.1 in chapter 2).

This chapter focuses on the first step in the ZMathLang approach to translating formal specifcations into theorem provers. The user can check for grammatical correctness with the aim to translate the specification fully into a theorem prover or they can use this step on it's own to check their specification for some sort of correctness.

The first part of this chapter explains the design of the ZCGa and how the rules and categories have been changed. It gives some examples of each of the categories and how they are used in Z. We then explain the rules in which the categories must follow in order to be ZCGa correct. The next section hgihlights some properties we can show about the ZCGa. Then we explain how the categories of the ZCGa syntax are adapted into weak types and check Z specification for correctness.

The final section demonstrates the implementation of the ZCGa and gives examples of certain errors one can get when checking a specification for grammatical correctness.

4.1 Weak Types

Since formal notation is a subset of mathematics we are able to adapt the CGa for mathematics to work for formal specification and thus the Z notation.

In order to check for grammatical correctness we introduce a weak type system for Z specifications illustrated in figures 4.1 and 4.2.

The ZCGa starts from it's lowest level, the *atomic level*, which underlines the elementary characters from which the syntax is made. It then builds itself up to the highest level, *discourse level* where the largest elements can be found. Everything in the *discourse* can be made from elements in the smaller levels. Everything in the *sentence level* can be made from the levels before and so on. Types in Z are not the same as weak types. Therefore we shall name each of the weak types, categories to eliminate confusion.

level	Main category	syntax	Meta-symbol
atomic	*variables*	$V = V^{\mathcal{T}} \mid V^{\mathbb{S}}$	x
	constants	$C = C^{\mathcal{T}} \mid C^{\mathbb{S}} \mid C^{\mathcal{E}}$	c
	binders	$B = B^{\mathbb{S}} \mid B^{\mathcal{E}}$	b
phrase	*terms*	$\mathcal{T} = C^{\mathcal{T}}(\overrightarrow{\mathcal{P}}) \mid V^{\mathcal{T}}$	t
	sets	$\mathbb{S} = \mathrm{C}^{\mathbb{S}}(\overrightarrow{\mathcal{P}}) \mid B^{\mathbb{S}}_{\mathcal{Z}}(\mathcal{E}) \mid V^{\mathbb{S}}$	s
sentence	*expressions*	$\mathcal{E} = C^{\mathcal{E}}(\overrightarrow{\mathcal{P}}) \mid B^{\mathcal{E}}_{\mathcal{Z}}(\mathcal{E})$	E
	definitions	$\mathcal{D} = C^{\mathbb{S}}(\overrightarrow{V}) := \mathbb{S}$	D
discourse	*schematext*	$\mathbf{\Gamma} = \emptyset \mid \mathbf{\Gamma}, \mathcal{Z} \mid \mathbf{\Gamma}, \mathcal{E}$	Γ
	paragraphs	$\Theta = \mathbf{\Gamma} \triangleright \mathcal{E} \mid \mathbf{\Gamma} \triangleright \mathcal{D}$	θ
	specifications	$\mathbf{Spec} = \emptyset \mid \mathbf{Spec}, \Theta$	spec

Other Category	abstract syntax	Meta-symbol
parameters	$\mathcal{P} = \mathcal{T} \mid \mathbb{S} \mid \mathcal{E}$	P
declarations	$\mathcal{Z} = \mathrm{V}^{\mathbb{S}}$:SET | $\mathrm{V}^{\mathcal{T}}$:$\mathbb{S}$	Z

Note: $\overrightarrow{\mathcal{P}}$ is a list of 0 or more $\mathcal{P}$'s, $\overrightarrow{\mathbb{S}}$ is a list of 0 or more $\mathbb{S}'s$, $\overrightarrow{\mathcal{E}}$ is a list of 1 or more $\mathcal{E}'s$, $\overrightarrow{V}$ is a list of 0 or more $V's$.

Table 4.1: Categories of ZCGa syntax.

These categories are adapted from the weak types in [40]. In particular *book* becomes *specification*, *lines* become *paragraphs*, *context* becomes *schematext* and *statements* become *expressions*. We eliminate *nouns*, *adjectives* and only have one syntax for *definition*.

4.1.1 Examples of specifications and weak types

Everything within a Z specification can be labelled using the categories found in table 4.1.

$$
\begin{array}{|l}
\hline
M \\
\hline
m : \mathbb{N} \\
n : \mathbb{P}\,\mathbb{N} \\
\hline
m \geq 0 \\
\#n \geq 1 \\
\hline
\end{array}
\qquad
\begin{array}{|l}
\hline
M' \\
\hline
M \\
m' : \mathbb{N} \\
n' : \mathbb{P}\,\mathbb{N} \\
\hline
m' = 0 \\
n' = \{\} \\
\hline
\end{array}
$$

Figure 4.1: Basic example of a specification

Using figure 4.1 we will give examples of individual weak type categories.

variables The set of variables V is divided into two subsets $V^{\mathcal{T}}$ and $V^{\mathbb{S}}$ which correspond to variables giving terms and variables giving sets respectively.

- $V^{\mathcal{T}}$. An example of a variable giving a term would be 'm' in figure 4.1.
- $V^{\mathbb{S}}$. An example of a variable giving a set would be 'n' in figure 4.1.

constants The set of constants range over constants giving terms $C^{\mathcal{T}}$, constants giving sets $C^{\mathbb{S}}$ and constants giving expressions $C^{\mathcal{E}}$.

- $C^{\mathcal{T}}$. An example of a constant giving an expression would be '0' in figure 4.1.
- $C^{\mathbb{S}}$. An example of a constant giving an expression would be '{}' in figure 4.1.
- $C^{\mathcal{E}}$. An example of a constant giving an expression would be '$m' = 0$' where the constants is '=' from figure 4.1.

binders There are two subsets of binders in the categories for Z specifications. Binders giving sets $B^{\mathbb{S}}$, and binders giving expressions $B^{\mathcal{E}}$.

- $B^{\mathbb{S}}$. An example of a binder giving an expression is
  '$\exists\, schedule : TIMESLOT \leftrightarrow ROOM \bullet (allPairsmoduleTT \cap schedule = \emptyset \land moduleTT = moduleTT \oplus m? \mapsto schedule)$'
  taken from Timetable specification in appendix **??**.

- $B^{\mathcal{E}}$. An example of a binder giving a set is
  '$\bigcup\{s : \mathrm{dom}\, studentTT \bullet \{s \mapsto (studentTT\, s \setminus moduleTT\, m?)\}$'
  taken from Timetable specification in appendix **??**.

terms Terms can range over constants giving terms with optional parameters $C^{\mathcal{T}}(\overrightarrow{\mathcal{P}})$, and variables giving terms $V^{\mathcal{T}}$.

- $C^{\mathcal{T}}(\overrightarrow{\mathcal{P}})$. An example of a constant giving a term is '$\#n$' (taken from figure 4.1) the constant being # which would be in the preface of constants with the weak typing as $\mathbb{S} \rightarrow \mathcal{T}$ and the parameter of this constant giving a term would be 'n' which is set.

- $V^{\mathcal{T}}$. See section 4.1.1 on variables giving terms.

sets The category of *set* has three sub categories, constants giving sets with optional parameters $C^{\mathbb{S}}(\overrightarrow{\mathcal{P}})$, binders giving sets with expression as its parameter $B^{\mathbb{S}}_{\mathcal{Z}}(\mathcal{E})$ and variables giving sets $V^{\mathbb{S}}$.

- $C^{\mathbb{S}}(\overrightarrow{\mathcal{P}})$. An example of a constant giving a set with parameters is
  '$studentTT = studentTT \cup s? \mapsto \emptyset$'
  taken from Timetable specification in appendix **??**. Where the constant giving a set is '$\cup$' and the parameters it takes is '$studentTT$' and '$s? \mapsto \emptyset$'.

- $B^{\mathbb{S}}_{\mathcal{Z}}(\mathcal{E})$. An example of a binder giving a set with an expression and declaration as parameters is

'$\bigcup\{s : \mathrm{dom}\, studentTT \bullet \{s \mapsto (studentTT\ s \setminus moduleTT\ m?)\}\}$'

taken from Timetable specification in appendix **??**. Where the constant is '$\bigcup$', the declaration parameter is '$s : \mathrm{dom}\, studentTT$' and the expression parameter is '$\{s \mapsto (studentTT\ s \setminus moduleTT\ m?)\}$'.

- $V^{\mathbb{S}}$. See section 4.1.1 on variables giving sets.

expressions The category of expressions ranges over two subsets, constants giving expressions with optional parameteres $C^{\mathcal{E}}(\overrightarrow{\mathcal{P}})$, and binders giving expressions with a declarations and expression $B^{\mathcal{E}}_{\mathcal{Z}}(\mathcal{E})$.

- $C^{\mathcal{E}}(\overrightarrow{\mathcal{P}})$. A constant giving an expression can be seen in figure 4.1 as '$m \geq 0$' where '$\geq$' is the constant giving an expression and the paramters are two terms: 'm' and '0'.

- $B^{\mathcal{E}}_{\mathcal{Z}}(\mathcal{E})$. A binder giving an expression could be a '$\forall$' or '$\exists$' binder. An example of this is shown in the Timetable specification in appendix **??** as

  '$\exists\, schedule : TIMESLOT \leftrightarrow ROOM \bullet (allPairsmoduleTT \cap schedule = \emptyset \land moduleTT = moduleTT \oplus \{m? \mapsto schedule\})$'

  where the binder giving an expression is '$\exists$', the declaration paramter is '$TIMESLOT \leftrightarrow ROOM$' and the binding expression is '$(allPairsmoduleTT \cap schedule = \emptyset \land moduleTT = moduleTT \oplus \{m? \mapsto schedule\})$'.

definitions Theres is only one kind of definition in the weak type theory syntax for Z. A local definition in Z is a constant giving a definitions taking variables as paramters giving a set, $C^{\mathbb{S}}(\overrightarrow{V}) := \mathbb{S}$. An example of this is shown the GenDB specification in appendix **??**. The definition is

'**let** $cosrel == (parent^{nth?+1} \fatsemi (parent^{-1})^{nth?+1+rem?}) \setminus (parent \fatsemi parent^{-1})\bullet$

$cousins! = cosrel(\!|\ \{p?\}\ |\!) \cup cosrel^{-1}(\!|\ \{p?\}\ |\!)$'

where the defined constant is *cosrel*.

schematext The schematext within a Z specification reighns over three sub categories, either the schema text can be empty $\emptyset$, or it can be schematext with a declaration $\mathbf{\Gamma}, \mathcal{Z}$ or it can be schematext with an expression $\mathbf{\Gamma}, \mathcal{E}$.

- $\emptyset$. The empty schema text is the beginning of a specification where we start with nothing.

- $\mathbf{\Gamma}, \mathcal{Z}$. The first declaration in a specification would be the empty Γ plus the declaration. For example in figure 4.1 the first example of this would be '$m : \mathbb{N}$', which is the empty schematext $\emptyset$ along with the first declaration of the specification. The second declaration add to the schema text would be $n : \mathbb{P}\,\mathbb{N}$ and so on.

- $\mathbf{\Gamma}, \mathcal{E}$. This set represents all the expressions which are added to the schema text. In the example in figure 4.1 we would already have two declarations in the schematext $m : \mathbb{N}$ and $n : \mathbb{P}\,\mathbb{N}$ in the schema text before the first expression is added $m \geq 0$. The second expression added to the schematext in the same example would be $\#n \geq 1$.

paragraphs A paragraph Θ contains either an expression $\mathbf{\Gamma} \triangleright \mathcal{E}$ or a definition $\mathbf{\Gamma} \triangleright \mathcal{D}$, relative to a schematext.

The symbol $\triangleright$ is a sepatation marker betwen the schematext and expression or definition

- $\mathbf{\Gamma} \triangleright \mathcal{E}$. Examples of expressions in a paragraph see section 4.1.1.

- $\mathbf{\Gamma} \triangleright \mathcal{D}$. Example of definitions in a paragraph see section 4.1.1.

specifications A specifiation *spec* is a list of paragraphs: $spec = \emptyset | \mathbf{Spec}, \Theta$.

A simple example of a specification is the entire of figure 4.1.

other categories Here we describe the other categories which are needed in the ZCGa abstract syntax.

Declarations. A declaration ina schematext represents the *introduction of a vairable* of a known type. In the categories of ZCGa syntax we can have two different kinds of declarations. This can be: `SET` (the type of all sets) or a set. Both of these declarations relates a *subject* (the left hand side of the declaration) with its *type/predicate*

(right hand side of the declarations). The abstract syntax for the two categories of declarations are $V^{\mathbb{S}}$ is a set $V^{\mathbb{S}} : SET$, or term $V^{\mathcal{T}}$ is in the set $\mathbb{S}$, $V^{\mathcal{T}:\mathbb{S}}$.

- $V^{\mathbb{S}} : SET$. An example of this kind of declaration is '$n' : \mathbb{P}\,\mathbb{N}$' taken from figure 4.1.
- $V^{\mathcal{T}:\mathbb{S}}$. An example of this kind of declaration is '$m : \mathbb{N}$' taken from figure 4.1.

Parameters. The list of paramters represent the categories in which constants may depend. The paramters available in the ZCGa abstract syntax are terms $\mathcal{T}$, sets $\mathbb{S}$ or expressions $\mathcal{E}$. For details on how each of these parameters are formed see the relevent sections (4.1.1, 4.1.1 and 4.1.1 respectively).

4.1.2 Weak Typing Rules

The ZCGa uses the weak types found in table 4.1 and checks the specification is correct according to the rules found in 4.2. To write the rules for the ZCGa we must first establish some definitions. The following definitions have been adapted from [40] to accomadate Z specifications.

Definition 4.1.1. *We abbreviate* $\vdash$ *spec* :: **Spec**, *spec* $\vdash \Gamma$:: $\mathbf{\Gamma}$ *as OK(spec;* Γ*)*

Definition 4.1.2. *The set dvar is the set of declared variables in the* schematext Γ*:*

- *If* $\Gamma = \emptyset$, *then* $dvar(\Gamma) = \emptyset$.
- *If* $\Gamma' = \Gamma, x : A$ *and* $x \notin dvar(\Gamma)$, *then* $dvar(\Gamma') = dvar(\Gamma), x$.
- *Else if* $\Gamma' = \Gamma, S$, *then* $dvar(\Gamma') = dvar(\Gamma)$.

Definition 4.1.3. *We denote a preface for a Z specification spec by prefcons(spec)* [1]. *This set contains all the constants listed in the preface. If* $c \in prefcons(spec)$ *and if* $K_1, ..., K_n$ *is the set of the weak types of the parameters of* c *and if* k *is the resulting weak type of the full construct* $c(...)$, *then we attatch the type* $k_1 \times ... \times k_n \rightarrow l$ *to* c.

Example 4.1.1.

[1] The full set of *prefcons* can be found in the implementation of the ZCGa checker. They are in under the variable *preface_constants*.

An example of a preface for a Z specification is shown in the following:

constant name	weak type	constant name	weak type
$\mathbb{N}$	$\mathbb{S}$	$<$	$\mathcal{T} \times \mathcal{T} \to \mathcal{E}$
$-$	$\mathcal{T} \times \mathcal{T} \to \mathcal{T}$	$\cup$	$\mathbb{S} \times \mathbb{S} \to \mathbb{S}$

Definition 4.1.4. *We denote a set containing all the constants to be defined in a specification spec by defcons(spec). Let $\Theta \in spec$ be a paragraph containing a definition $\Gamma \triangleright \mathcal{D}$ where $\mathcal{D}$ is in the form $c(x_1, ..., c_n) := A$. Then the defined constant of the definition, or defcons($\mathcal{D}$), is c.*

(var)	$\dfrac{OK(spec;\ \Gamma), x \in V^{\mathcal{T}/\mathbb{S}}, x \in dvar(\Gamma)}{spec;\ \Gamma \vdash x :: \mathcal{T}/\mathbb{S}}$
(int-cons)	$\dfrac{\begin{array}{r} OK(spec;\ \Gamma), \Gamma' \triangleright \mathcal{D} \in spec, \\ dvar(\Gamma') = \{x_1, ..., x_n\}, defcons(\mathcal{D}) = c \in C^{\mathcal{T}/\mathbb{S}/\mathcal{E}}, \\ wt_{spec;\,\Gamma}(P_i) = wt_{spec;\,\Gamma'}(x_i),\ \text{for all } i = 1, ..., n \end{array}}{spec;\ \Gamma \vdash c(P_1, ..., P_n) :: \mathcal{T}/\mathbb{S}/\mathcal{E}}$
(ext-cons)	$\dfrac{\begin{array}{r} OK(spec;\ \Gamma), c \text{ external to spec}, c :: k_1 \times ... \times k_n \to k, \\ spec;\ \Gamma \vdash P_i :: k_i (i = 1, ..., n) \end{array}}{spec;\ \Gamma \vdash c(P_1, ..., P_n) :: k}$
(bind)	$\dfrac{OK(spec;\ \Gamma;\ Z), b \in B, b :: k_1 \to k_2, spec;\ \Gamma, Z \vdash E :: k_1}{spec;\ \Gamma \vdash b_z(E) :: k_2}$
(defin)	$\dfrac{\begin{array}{r} spec;\ \Gamma \vdash s :: \mathbb{S} \\ dvar(\Gamma) = \{x_1, ..., x_n\}, c \in C^{\mathbb{S}}, c \notin prefcons(spec) \cup defcons(spec) \end{array}}{spec;\ \Gamma \vdash c(x_1, ..., x_n) := s :: \mathcal{D}}$
	$\dfrac{\vdash spec :: \mathbf{Spec}}{spec \vdash \emptyset :: \mathbf{\Gamma}} (emp - cont)$
(set-dec)	$\dfrac{OK(spec;\ \Gamma), x \in V^{\mathbb{S}}, x \notin dvar(\Gamma)}{spec \vdash \Gamma, x : SET :: \mathbf{\Gamma}}$
(term-dec)	$\dfrac{OK(spec;\ \Gamma), spec;\ \Gamma \vdash s :: \mathbb{S}, x \in V^{\mathcal{T}}, x \notin dvar(\Gamma)}{spec \vdash \Gamma, x : s :: \mathbf{\Gamma}}$
(assump)	$\dfrac{OK(spec;\ \Gamma), spec;\ \Gamma \vdash e :: \mathcal{E}}{spec \vdash \Gamma, e :: \mathbf{\Gamma}}$
(emp-spec)	$\dfrac{}{\vdash \emptyset :: \mathbf{Spec}}$
(spec-ext)	$\dfrac{spec \vdash \Gamma :: \mathbf{\Gamma}}{\vdash spec, \Gamma :: \mathbf{Spec}}$

Table 4.2: Weak typing rules used by the ZCGa type checker.

4.1.3 Weak typing properties and definitions

Since the categories and rules of the Z syntax WT_Z are a subset of the original MathLang WT_M [40], the following lemma holds:

Lemma 4.1.1. *ZCGa properties*

1. *Types are unique. I.e. if* $spec, \Gamma \vdash E :: W$ *then* W *is unique.*
2. *Type finding is decidable. I.e. for any* $spec, \Gamma, E$*, we can decide if there is* $W /$ $spec, \Gamma \vdash E :: W$*.*
3. *Type checking is decidable. I.e. for* $spec, \Gamma, E$, W *then we can decide if* $spec, \Gamma \vdash E :: W$*.*

4.1.4 Adapting weak types to the ZCGa

For our ZCGa checker we take the core categories from table 4.2.

We use 7 categories, **Spec**, $\mathbf{\Gamma}$, $\mathcal{T}, \mathbb{S}, \mathcal{Z}, \mathcal{E}, \mathcal{D}$ corresponding to *specification*, *schematext*, *term*, *set*, *declaration*, *expression*, and *definition* respectively. These categories and weak typing rules will aid us to translate a specification into a full proof as they help us complete the GPSa (see Figure 3.1).

4.2 Annotations

Using the ZMathLang LaTeX package the user can label the specification with ZCGa annotations. This can be either before or after labelling the specification with ZDRa (see chapter **??**). The ZCGa annotations will highlight each individual grammatical aspect of the specification. Table 4.3 shows how to label specifications with ZCGa.

Category	LaTeX label	Colour
Specification	`\specification{...}`	
SchemaText	`\text{...}`	
Term	`\term{...}`	
Set	`\set{...}`	
Declaration	`\declaration{...}`	
Expression	`\expression{...}`	
Definition	`\definition{...}`	

Table 4.3: ZCGa LaTeX annotations and their colours.

4.2.1 term

According to the rules in table 4.2 for an element to be a well typed term it can be a variable being declared such as t (labelled in blue) in figure 4.4 or it can be a constant giving a term. The latter kind of term must have a constant within the preface of constants and a variable which has been declared. An example of this could be the term '`# s`' (shown in figure 4.2).

`\term{\# \set{s}}`	$\#\,s$

Figure 4.2: Constant giving a term

Figure 4.2 shows a constant giving a term. Provided that the set s is in the set of declared variable then the term `# s` is a correctly typed term.

4.2.2 set

Similar to typing a term, set can be correctly type in one of two ways. The first is a variable set which is correctly declared such as 's' in figure 4.5. The second way a set could be correctly typed is by having a constant with parameters.

`\set{\set{s} \cup \set{s'}}`	$s \cup s'$

Figure 4.3: Constant giving a term

Figure 4.3 shows an example of a correctly typed set, consisting of a constant and in this case two parameters (s and s'). So long as s and s' are in the set of declared variables then '$s \cup s'$' is a correctly typed set.

4.2.3 declaration

There are two types of grammatically correct declaration:

1. term declaration
2. set declaration

A term declaration is any declaration, expressing the relation between something and it's type. For example if we had the declaration `t: \nat` this is declaring t is of type some sort of natural number. We can label this declaration in ZCGa (shown in figure 4.4)

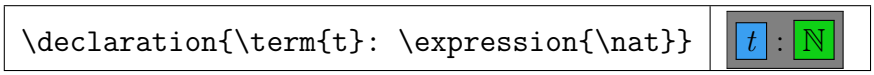


Figure 4.4: Correct term declaration labelled in zcga

We label $\mathbb{N}$ as declarations for the same reasons as the typing's behaviour in [40].

The second type of declaration would be the declaration of a set, for example `s: \power \nat` this is saying that the set s is in the set $\mathbb{P}\,\mathbb{N}$. Figure 4.5 shows how this kind of declaration would be labelled in ZCGa

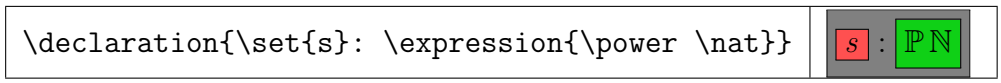


Figure 4.5: Correct set declaration labelled in zcga

4.2.4 expression

An expression (named assump in the rules) is any correct expression within the context. The expression can only contain correct sets and terms in the context. Any constants within the expression must be in the preface of the ZCGa checker. A correct expression is shown in figure 4.6.

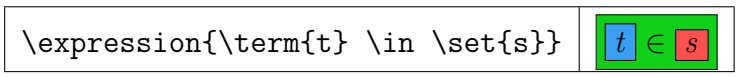


Figure 4.6: Correct expression labelled in zcga

4.2.5 definition

If all the variables within the definition have been declared and the constant the user is defining is a constant set. As well as the constant the user is defining is not already in predefined and preface constants then the definition is correct. A correct Z definition is shown in figure 4.7.

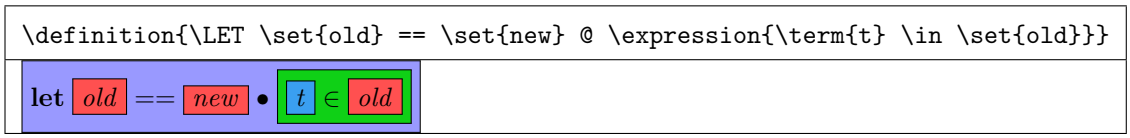


Figure 4.7: Correct definition labelled in zcga

4.2.6 schematext

Schematext is all the correct declarations, expressions and definitions within a specification. Figure 4.8 shows schematext which is correct. The declaration and expression within the content of the schema would be classified as schematext.

```
\begin{schema}{K}
\text{\declaration{\set{s}:
\expression{\power \nat}}}
\where
\text{\expression{\term{t} \in \set{s}}}
\end{schema}
```

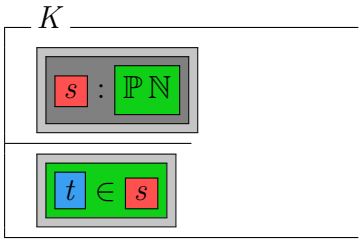


Figure 4.8: Example of correct schematext

4.2.7 specification

Specifications are correct when the schematext is empty or all the schematext within a specification is correct. A full example of a correctly labeled specification in ZCGa is shown in chapter 10 in figure 10.4.

4.3 Implementation

The ZCGa program automtically checks the specification for grammatical correctness. To do this it uses the ZCGa annotions inputted by the user and will notify the user if all is correct or any errors it may encounter. The ZCGa checker is a weak type checker. Therefore it only checks the correctness of the weak types of the specification (annotated) and not actual Z types like a Z type checker would find, such as fuzz.

4.3.1 Checking if a specification is ZCGa correct

The ZCGa program uses regular expressions to read the annotations written by the user to determine with a specification is ZCGa correct. It ignores all the parts which are not labelled in ZCGa which allows for semi-formal specification to be checked as well. Look at the following:

Example 4.3.1. `There is a variable t which is a number that:`

```
\text{\expression{\term{j} < \term{k}}}
```

This example shows a small specification that is semi-formal specification, the ZCGa will read the annotations `\text`, `\expression` and `\term`. It will see that the specification contains a correct expression, however the specification is ZCGa incorrect as it would not pick up that the terms have been declared. For this example to be ZCGa correct the user will need to change it into the example shown in 4.3.2.

If the specification is correctly labelled and follows all the rules in table 4.2 then a message would appear saying `Spec Grammatically Correct`. However if the specification is ZCGa incorrect then a message will appear saying '`Spec Grammatically Incorrect, Number of errors:`n' where n will be a number with the number of errors.

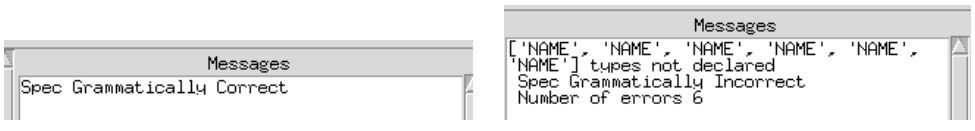


Figure 4.9: Message shown when specification is correct (left) and incorrect (right).

In figure 4.9 the left image shows the message when a specification is ZCGa correct. The right image shows a specification in which a specification is not ZCGa correct as the type '`NAME`' has been used 6 times in the specification but has not been declared.

4.3.2 Errors

The specification is ZCGa correct when all labelled objects within the specification follow the ZCGa type rules (table 4.2). Here we highlight what error messages one might get when running the ZCGa checker on a specification.

term not declared This follows the rules for variables in table 4.2. The message '*term not declared*' will appear if a term is labelled within the schematext and there hasn't been a declaration defining its type previously. Take a look at the following example fo a specification:

Example 4.3.2.
```
\begin{schema}{K}
\text{\declaration{\term{j}:\expression{\nat}}}
\where
\text{\expression{\term{j} < \term{k}}}
\end{schema}
```

In the schematext containing the expression `\term{j} < \term{k}` the term `j` has been previosuly declared with the type `\nat` however the term `k` is labelled and used in the expression however it has not been declared or assigned a type. This will cause the error message *term not declared*.

set not declared Similar to the previous error message, this follows the rules for variables in table 4.2. The message *set not declared* will appear when a variable is labelled `\set{..}` in the schematext of a specifiation but has not been declared previously. Look at the following example:

Example 4.3.3.
```
\begin{schema}{U}
\text{\declaration{\term{j}:\expression{\nat}}}
```

```
\where
\text{\expression{\term{j} \in \set{js}}}
\end{schema}
```

When the ZCGa checker runs through this specification the error message *js: set not declared* should appear. This is because the term `j` has been declared, however the set `js` has not been declared yet it is used in the schematext `\term{j} \in \set{js}`.

constant not in preface When a constant is used within a specification that is not in the preface (see ZCGa code to see all constants in preface) then the '*constant not in preface*' error message will appear. An example is shown in the following specification:

Example 4.3.4.
```
\begin{schema}{T}
\text{\declaration{\set{t}:\expression{\mathbb{P} \nat}}}
\where
\text{\expression{\set{t} = \set{\{\}}}}
\end{schema}
```

The error message will appear here when running the ZCGa checker on the specification as the constant '`\mathbb{P}`' is not in the preface. The user in this case may have meant to use the constant `\power` instead of `\mathbb{P}`. Even though these two constants look identical when compiling a LaTeX document they are not the same when checking specifications with ZCGa.

constant already in specification The error message *constant already in specification* will appear if the user tries to define a constant which is already in the preface constants or defined constants. For example take a look at the following specification:

Example 4.3.5.
```
\set{\nat} := \term{1} | \term{2} | \term{3} |...
```

The ZCGa would say this specification is incorrect and the error '*constant already in specification*' would appear as the user is trying to define the set of natural num-

bers \nat however this constant is already programmed in the preface of constants for Z specifications.

This error would also appear if the user tried to define a constant such as [STUDETS] more than once in their specification.

not a correct term For this error message to appear, the user must have tried to create a term when it wasn't allowed. Take the following example:

Example 4.3.6.
```
\begin{schema}{Y}
\text{\declaration{\term{y}:\expression{\nat}}}
\where
\text{\expression{\term{\# \term{y}} = \term{0}}}
\end{schema}
```

This specification would be ZCGa incorrect and the error message *not a correct term* would appear due to the term \term{\# \term{y}} being incorrect. This is because the constant \# takes a set as a parameter and gives back a term (the cardinality of the set). In this case the user has applied a term \term{y} to the constant \# and thus the error message appearing.

not a correct set The error message *not a correct set* will appear when a user has labelled something as a set when it is not. For example take the following specification:

Example 4.3.7.
```
\begin{schema}{W}
\text{\declaration{\term{w}:\expression{\power \nat}}}
\text{\declaration{\term{w'}:\expression{\power \nat}}}
\text{\declaration{\term{v}:\expression{\nat}}}
\where
\text{\expression{\set{w'} = \set{\set{w} \cup \term{v}}}}
\end{schema}
```

In this case the incorrect set would be \set{\set{w} \cup \term{v}}. This is because the constant \cup takes two sets as parameters. However this labelling

shows that a set '`w`' and a term '`v`' have been applied.

An example of a specification not passing the ZCGa check is shown in appendix **??**. This specification shows a telephone directory which adds telephone numbers and names to a theoretical directory. It shows a schema '*TheTelephoneDirectory*' being usedd in the *AddPerson* schema, however '*TheTelephoneDirectory*' is not a valid schema. The term '*person*' is being used in the '*AlreadyInDirectory*' schema but only the variable '*persons*' has been declared. The term '*numberInUse*' is being used in the '*NameNotInDirectory*' schema however the user might of mistaken this term for the '*nameNotInDirectory*' which has been declared in the '$OUTPUT$' freetype. The user has also forgotten the variable decorations '*?*' and '*!*' on the 'n' and 's' variables in the '*AddNumber*' schema.

All these errors may not be visible to the user when typing the specification or looking at it with the naked eye, however with the user going through and labelling each part in ZCGa annotations along with the ZCGa checker, all these grammatical errors should be identified.

4.4 Benefits

In addition to the main use for the ZCGa checker which is to check the grammar of a formal specification written in Z other [2] benefits exists. The ZCGa would also be an advantage to the user or designer of a system to translate their ideas to the developers of the system. For example by using the ZCGa the developers can clearly see which parts of the systems are represented as sets and which parts are represented as terms. Not only does it help describe the system to developers but other memebers of the project development team and other stakeholders such as the client would also get a better idea of the layout of the system. A further advantage to the ZCGa is that it is able to check the grammatical correctness of partially-formal specifications. These can include specifications written in the english natural langugage but are on their way to becoming formal, or specifications with formal

[2] As a side note, when copying specifications into this thesis I mistyped some words and they were only caught when I ran the ZCGa checker on them

parts to them.

4.5 ZCGa on a semiformal specification

The ZCGa can also be used on semiformal specification. An example is shown in appendix A.5, which describes an auto pilot system. This specification is written partly in the english natural language and partly in Z. Therefore the specification is on it's way to becoming formal. The user can then annotate the formal parts and some of the informal parts in ZCGa, it can then be checked and will the user if there are any grammatica errors in the specification so far, and if any variables which have been used need to be declared etc. For example, in the auto pilot specification, the `off_eng` schema has a declaration which states `mode:mode_status` (figure 4.10). If the `mode_status` type was not declared before it was used in the `off_eng` schema, then the ZCGa checker would identify this and would display an error message. However, after the type `mode_status` has been declared in the specification it can be used throughout the rest of the specification, including in the informal text part.

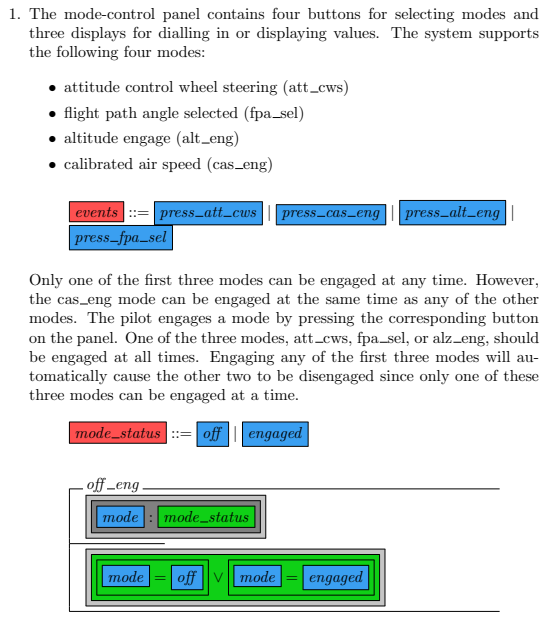


Figure 4.10: Part of the Autopilot specification labelled in ZCGa.

The full version of the semi formal specification is shown in appendix A.5.3.

Even though this specification is only partially formal, we are able to translate the annotated parts all the way to the Isabelle syntax (step 5 from figure 3.1).

4.6 Conclusion

In this chapter we have seen how the ZCGa has grown from weak type theory for mathematics [40]. We have giving examples of different categories are used within a Z specification and highlighted the rules these categories need to follow in order to be ZCGa correct. We have described a few properties of the checker and have explained how these categories are transformed into weak types for Z. We explained how a Z specification can be annotated and checked bu the ZCGa and given and illustrated the different errors which may arise. The next step of the ZMathLang framework to check for another type of correctness, the ZDRa.

Chapter 5

Z Document Rhetorical aspect

The ZDRa is similar to the DRa for mathematics. Here we describe how the ZDRa was designed an implemented.

We use the ZMathLang LaTeX package to chunk specifications together and see the relationships between them. The mathematical instances used are *theory* and *axiom*, which are used in theorem prover syntax. We also use *precondition*, *post-condition*, *output*, *stateInvariants*, *stateschema*, *outputschema*, *changeschema* and *totaliseSchema*.

We created the ZDRa for the following:

- Identifying loops in the reasoning of specifications.
- Checking the specification is robust by making sure schemas have been totalised.
- Identifying the relationships between chunks of specification.
- Making sure state invariants do not change throughout the specification.
- Creating a dependency graph to create a formal proof sketch from the ZDRa.

As well as having instances, the ZDRa shows relations between them to make sure there are no loops in reasoning and to give warning in some situations (e.g. specification is not totalised).

Section 5.1 describes the labels used to annotate a specification. Then in section 5.2 we illustrate the implementations of the ZDRa and how to check for rhetorical correctness. The rhetorical errors which can be found are explained in sections 5.2.2.1 and 5.2.1.1 and the products which are created when a specification is rhetorically correct are described in section 5.2.3.

5.1 Annotations

Using our ZMathLang LaTeX package we can label the specifications with ZDRa annotations (either before of after the ZCGa). These annotations chunks parts of the specification together and upon compiling shows the relationships between each of these chunks.

Instance	**Notation**	**LaTeX Command**
theory	T	\dratheory{T}{$scaleoftheory$}{$instance$}
stateschema	SS	\draschema{$SS\#$}{$instance$}
initschema	IS	\draschema{$IS\#$}{$instance$}
changeschema	CS	\draschema{$CS\#$}{$instance$}
outputschema	OS	\draschema{$OS\#$}{$instance$}
totalise	TS	\draschema{$TS\#$}{$instance$}
axiom	A	\draschema{$A\#$}{$instance$}
stateInvariants	SI	\draline{$SI\#$}{$instance$}
precondition	PRE	\draline{$PRE\#$}{$instance$}
postcondition	PO	\draline{$PO\#$}{$instance$}
output	O	\draline{$O\#$}{$instance$}

Table 5.1: ZDRa instances with their notations and LaTeX commands.

Relation	LaTeX Command
initialOf	\initialof {*instance_1*}{*instance_2*}
uses	\uses {*instance_1*}{*instance_2*}
requires	\requires {*instance_1*}{*instance_2*}
allows	\allows {*instance_1*}{*instance_2*}

Table 5.2: ZDRa Relations with their notations and LaTeX commands.

Table 5.1 shows the type of instance available in a Z specification, the notation that goes along with it and the LaTeX command the user annotates that part of the specification with. It is important to name instances as these names are what are referred to when creating the relationships between them. Table 5.2 shows the relationships which are available between some of the instances.

5.1.1 Instances

We have designed the notation to include `\draschema{..}{..}` for chunks of specification which include schemas and `\draline{..}{..}` which only include lines within a schema.

5.1.1.1 theory

A *theory* would be a whole specification of one particular system. Appendix ?? shows an entire specification labelled in ZDRa. You can have more than one *theory* in a single document. The *theory* would contain all other instances within it but no instance can have a *theory* inside of it.

5.1.1.2 stateschema

A *stateschema* just like in Z is a single instance which outlines the state of the system. Figure 5.1 shows an example of a *stateschema* instance. Note we have the label `\draschema{SS1}{....}` (shown in red) where we have labelled this *stateschema* `SS1`. There may be one or more *stateschema*'s in a theory. So users can label their

stateschema's accordingly, e.g. `SS1, SS2, SS3`.....[1]

```
\draschema{SS1}{
\begin{schema}{BirthdayBook}
known: \power NAME \\
birthday: NAME \pfun DATE
\where
\draline{SI1}{known=\dom birthday}
\end{schema}}
```

Figure 5.1: A stateschema and stateinvariants labelled in ZDRa.

5.1.1.3 initialschema

An *initialschema* instance is optional within a *theory*, there can be more than one depending on how many *stateschema*'s there are. Figure 5.2 shows an example of an *initialschema*. We use the labelling `\draschema{IS1}{...}` (shown in red) to denote the *initialschema*. We have named the *initialschema* `IS1` to show it is the first initial schema in the *theory*. If there was another *initialschema* we would name it `IS2` and so on.

```
\draschema{IS1}{
\begin{schema}{InitBirthdayBook}
BirthdayBook'
\where
\draline{PO2}{known' = \{ \}}
\end{schema}
}
```

Figure 5.2: An initialschema labelled in ZDRa.

5.1.1.4 changeschema

A *changeschema* instance is a schema/function which changes the current state of the specification (these schemas are usually denoted by having a `\Delta` operator). There can be none or many *changeschema* instances within a theory.

[1] Although it is not needed for the user to annotate the instances incrementally (the instances just need different numbers) it makes it easier to identify how many of each instance is within the specification.

```
\draschema{CS1}{
\begin{schema}{AddBirthday}
\Delta BirthdayBook \\
name?: NAME \\
date?: DATE
\where
\draline{PRE1}{name? \notin known}
\draline{PO3}{birthday' = birthday \cup \{name? \mapsto date?\}}
\end{schema}
}
```

Figure 5.3: A changeschema labelled in ZDRa.

Figure 5.3 shows an example of a *changeschema* annotated in ZDRa. The schema is labelled with: `\draschema{CS1}{...}` (shown in red). We name this instance `CS1` as it is the first *changeschema* instance seen in the specification, the next *changeschema* should be named `CS2` then `CS3` etc.

5.1.1.5 outputschema

An *outputschema* instance is a schema or a function which does not change the current state but only outputs information from the current state (these schemas are usually denoted by having a `\Xi` operator). Figure 5.4 shows an example of an *outputschema* instance in ZDRa. Note the line `\draschema{OS4}{...` (shown in red) which names the chunk `OS4` that is the forth *outputschema* in the specification.

```
\draschema{OS4}{
\begin{schema}{AlreadyKnown}
\Xi BirthdayBook \\
name?: NAME \\
result!: REPORT
\where
\draline{PRE3}{name? \in known} \\
\draline{O4}{result! = already\_known}
\end{schema}
}
```

Figure 5.4: A outputschema labelled in ZDRa.

5.1.1.6 totalise

A *totalise* instance are parts of the specification which totalise preconditions and schemas within the specification. These are labelled as `TS#` where `#` is some number. *Totoalise* instances can written in two ways. Either using the double equals operator:

```
\begin{zed}
A == B \land C
\end{zed}
```

or by using the `defs operator`:

```
\begin{zed}
A \defs B \land C
\end{zed}
```

When labelling *totalise* instances the user can do this in two ways. The first labelling is as a `draschema` where the labelling comes before the `\begin{zed}` or as a `draline` where the labelling wraps around the line of the instance only. The second way is useful if there are more than one *totalise* instance between the `\begin{zed}` and `\end{zed}`.

```
\draschema{TS3}{
\begin{zed}
VM3 \defs VM1 \lor VM2
\end{zed}}
```

Figure 5.5: A totalise schema instance labelled in ZDRa.

An example of the `draschema` labelling is shown in figure 5.5 where we have the label `\draschema{TS3}{..` (shown in red).

```
\begin{zed}
\draline{TS1}{RAddBirthday == (AddBirthday \land Success)\\  \lor AlreadyKnown} \\
\draline{TS2}{RFindBirthday == (FindBirthday \land Success) \lor NotKnown} \\
\draline{TS3}{RRemind == Remind \land Success}
\end{zed}
```

Figure 5.6: A totalise line instance labelled in ZDRa.

An example of the `draline` instance is shown in figure 5.6. In this example we have three *totalise instances* using the label `\draline{TS1}{...`, `\draline{TS2}{...` and `\draline{TS3}{...` respectfully (shown in red).

Totalise instances become the properties to prove when converted to the half-baked proof (see section **??** for details).

5.1.1.7 axiom

Axiom instances are knowns as axiomatic definitions in Z. There can be more than one *axiom* in a *theory* or there can be none. An example of an *axiom* instance labelled in ZDRa is shown in figure 5.7. Note the ZDRa labelling consists if the line `\draschema{A1}{...` where the instance is named `A1`.

```
\draschema{A1}{
\begin{axdef}
maxPlayers: \nat
\where
maxPlayers = 20
\end{axdef}}
```

Figure 5.7: A axiom instance labelled in ZDRa.

5.1.1.8 stateInvariants

The *stateInvariants* instance are the conditions which must be obeyed throughout the specifications. These are the lines founds inside the *stateschema* instance. Figure 5.1 shows a single *stateinvariant* instance labelled as `\draline{SI1}{..` (shown in blue). There can be 0 or more *stateinvariance* instances within a *theory*.

5.1.1.9 precondition

Similar to the *totalise* instance, the *precondition* instance can be labelled as a `draschema` or a `draline`. An example of a *precondition* which is a line can be found in figures 5.3 and 5.4. In figure 5.3 the DRa labelling for a *precondition* schema is the line `\draline{PRE1}{..}` (shown in blue). The *precondition* instance is named `PRE1` and `name? \notin known` is the instance. In figure 5.4 the ZDRa labelling is `\draline{PRE3}{..}` (shown in blue) where `PRE3` is the name of the instance and `name? \in known` is the instance.

Another way a *precondition* instance can exists is when an entire schema only consist of *precondition* instances and nothing else (no post operations).

```
\draschema{PRE1}{
\begin{schema}{exact\_cash}
cash\_tendered?: \nat
\where
cash\_tendered? = price
\end{schema}}
```

Figure 5.8: A precondition schema instance labelled in ZDRa.

A *precondition* instance which is an entire schema is shown in figure 5.8. The ZDRa labelling consists of the line `\draschema{PRE1}{..` where the name of the instance is `PRE1`.

5.1.1.10 postcondition

The *postcondition* instance can be labelled as a ZDRa line. An example of this is demonstrated in figure 5.3 (shown in green), where the name of the instance is `PO3` and the instance itself is `birthday' = birthday \cup \{name? \mapsto date?\}`.

5.1.1.11 output

An *output* instance can also be labelled as a ZDRa line. An example of this is shown in green in figure 5.4. The instance in this case is named `O4` and the instance itself is `result! = already\_known`.

5.1.2 Relations

After labelling the instances within a relation the user may then add relations between parts of the specification. The relationships available are *initialOf*, *uses*, *requires* and *allows*.

requires			**uses**		
outputSchema	$\longrightarrow$	precondition	outputSchema	$\longrightarrow$	stateSchema
outputSchema	$\longrightarrow$	output	changeSchema	$\longrightarrow$	stateSchema
changeSchema	$\longrightarrow$	precondition	stateSchema	$\longrightarrow$	stateSchema
changeSchema	$\longrightarrow$	postcondition	stateSchema	$\longrightarrow$	axiom
totalises			outputSchema	$\longrightarrow$	axiom
totalise	$\longrightarrow$	changeSchema	changeSchema	$\longrightarrow$	axiom
totalise	$\longrightarrow$	outputSchema	**allows**		
totalise	$\longrightarrow$	totalise	precondition	$\longrightarrow$	postcondition
			initialOf		
			initialSchema	$\longrightarrow$	stateSchema

Table 5.3: The legal relations between instances. Where $\longrightarrow$ represents the relation.

Table 5.3 shows the legal relations between each of the instances for example an *initialSchema* can be an *initialOf* a *stateSchema* but it wont allow say a *changeSchema* to be *initialOf* an *initialSchema* etc.

5.1.2.1 initialOf

An *initialschema* instance can be an *initialOf* a *stateschema* instance. An example of this would be `\initialOf{IS1}{SS1}` where `IS1` is *initialOf* `SS1`.

5.1.2.2 uses

The *uses* relation can be between *changeSchema*'s, *outputSchema*'s, *stateSchema*'s and *totalise* instances. For example, one can say that an *outputSchema uses* a *stateSchema*. To illustrate this the user can add the label `\uses{OS2}{SS1}`, meaning *outputSchema* OS2, uses *stateSchema* SS1.

5.1.2.3 requires

The *requires* relation is used between an *outputSchema* or a *changeSchema* and a *precondition*, *output* or *postcondition*. If we take the example shown in figure 5.2 we can say that the *initialSchema* `IS1` *requires* the postcondition `PO2`. Therefore in ZDRa notation we would write in our LaTeX specification `\requires{IS1}{PO2}`.

5.1.2.4 allows

The *allows* relation is used between *precondition*'s and *postconditions* or *preconditions* and *outputs*. This relation describes if an instance allows another instance to occur. That is an instance can not exist unless pre-conditional requirements are met. An example of where an *allows* relationship would be useful is in figure 5.4, where we have a *precondition* `PRE3` (in blue) and an *output* `O4` (in green). In this case the user would write in their ZDRa LaTeX file: `\allows{PRE3}{O4}`.

5.2 Implementation

The ZDRa program automatically checks for rhetorical correctness of the specification. To do this it reads the ZDRa annotations created by the user then if all is correct then the program automatically generates a *dependency graph* and a *goto graph*. The input for the ZDRa program is the specification written in LaTeX with the ZDRa annotations.

5.2.1 Checking if a specification is correctly totalised

The ZDRa program uses regular expressions to read the annotations inputted by the user to determine whether a specification has been totalised. For example if we only have a single *precondition* instance (PRE1), in a specification and a single *totalise* instance (TS1), in a specification and the user has added the *relation* `totalises{TS1}{PRE1}`, then the specification will be correctly totalised and there will be a message saying "`Specification correctly totalised`". If there exists any preconditions which the user hasn't annotated with a totalising relation then the ZDRa program will display a message saying "`Specification not correctly totalised`". This message is a warning not an error therefore even if the specification still has untotalised preconditions the user can still go on with the next steps of computerisation.

5.2.1.1 Errors

Table 5.4 shows examples of 3 specifications (named 1, 2 and 3). Specification 1 shows three *preconditions* being annotated by the user (PRE1, PRE2 and PRE3), all these preconditions have been annotated with the relationship *totalises*. Therefore if all preconditions in a schema have a totalising condition then the specification is correctly totalised. Specification 2 in the table shows 4 existing preconditions. All but one (PRE2) precondition have a totalising relationship with a totalise schema. In this case, PRE2 is an outstanding preconditions to be totalised and therefore the message '`Specification incorectly totalised` appears. Specification 3 in table 5.4 shows 5 schema preconditions. When checking totalising correctness it does not

matter whether the preconditions are in `draline` form or `draschema` form. In the third example we see that PRE3 and PRE4 have not been totalised and again a message appears saying the specification has not been correctly totalised.

	Preconditions in specification	**Totalises in specification**	**Outcome of ZDRa program**

1	`\draline{PRE1}`
	`\draline{PRE2}`
	`\draline{PRE3}`
		`\totalises{TS1}{PRE1}`
		`\totalises{TS1}{PRE2}`
		`\totalises{TS2}{PRE3}`
			'Spec correctly totalised'

2	`\draline{PRE1}`
	`\draline{PRE2}`
	`\draline{PRE3}`
	`\draline{PRE4}`
		`\totalises{TS1}{PRE1}`
		
		`\totalises{TS2}{PRE3}`
		`\totalises{TS3}{PRE4}`
			'Spec incorrectly totalised'

3	`\draschema{PRE1}`
	`\draschema{PRE2}`
	`\draschema{PRE3}`
	`\draschema{PRE4}`
	`\draschema{PRE5}`
		`\totalises{TS1}{PRE1}`
		`\totalises{TS2}{PRE2}`
		
		`\totalises{TS2}{PRE5}`
			'Spec incorrectly totalised'

Table 5.4: Examples of preconditions in a specification being correctly totalised and incorrectly totalised.

5.2.2 Checking if a specification has no loops in it's reasoning

The ZDRa program also checks for rhetorical correctness, that is it checks that there are no loops in the logical reasoning of the specification. To do this the ZDRa program imports a module named `networkX` to create a *directed graph*. The program also uses *regular expressions* to read the ZDRa annotations and create nodes and edges. For example, if the program finds `\draline{x}{....}` or `draline{y}{...}` then it will add `x` and `y` as nodes to the directed graph. The edges are created by reading the *relations* in the ZDRa annotated specification. For example if the ZDRa

program finds `\uses{x}{y}` and `x` and `y` are nodes in the directed graph then it will add a directed edge between `x` and `y` respectively. Nodes with no edges at all can also be added to the graph.

5.2.2.1 Errors

The specification is correct when there are no loops in the created directed graph. For example if there was a graph with edges [a→b, a→c, b→c] then that would still be legal as there are no directed loops, however, if there was a graph with edges [a→b, b→c, c→a] then that would cause a loop in the reasoning and the specfication will not be ZDRa correct. The program outputs a message informing the user whethere the specification is ZDRa correct or not.

```
...
\draschema{SS1}{
\begin{schema}{A}
C
\end{schema}}
\draschema{SS2}{
\begin{schema}{B}
A
\end{schema}}
\draschema{SS3}{
\begin{schema}{C}
B
\end{schema}}

\uses{SS1}{SS3}
\uses{SS2}{SS1}
\uses{SS3}{SS2}
...
```

Figure 5.9: An example of a loop in the reasoning in a labelled ZDRa specification.

Figure 5.10: The pdflatex output of figure 5.9.

Figure 5.10 shows the relationship SS1 *uses* SS3, SS2 *uses* SS1 and SS3 *uses* SS2. The ZDRa would not allow this as the reasoning would be in a loop and would not be correct. When running the ZDRa check on this specification the message which would appear is shown in figure 5.11

```
Specification Correctly Totalised
 Error! Circular Reasoning:
 Path of loop: [['SS3', 'SS1', 'SS2']]
```

Figure 5.11: An example of an error message when a specification is not ZDRa correct.

If the specification is ZDRa correct then the program also creates visual dependency and GoTo graphs automatically (see section 5.2.3). If not then the graphs are not created.

A full specification which passes ZCGa is shown in appendix A.4. There is one loop in the reasoning of this specification and the user is able to see it when they annotated the specification in ZDRa and have compiled the document using `pdflatex`. A snippet of this is shown in figure 5.12 and the full version is shown in appendix A.4.2.

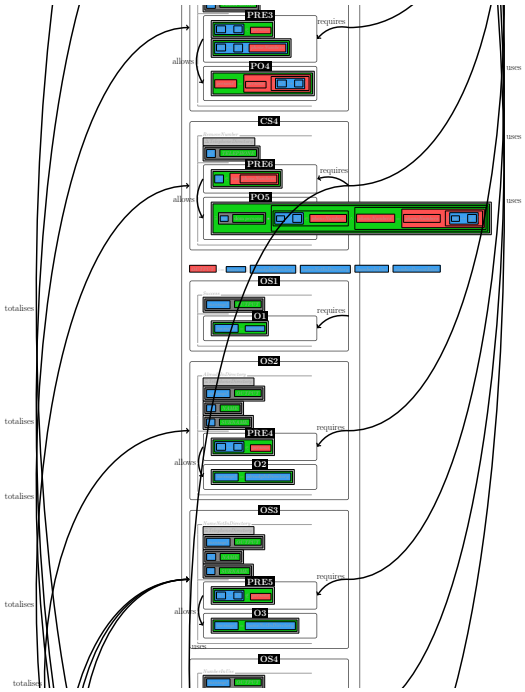


Figure 5.12: A snippet from appendix A.4 showing a loop in reasoning.

When the specification is checked with ZCGa then the output message says the specification is grammatically correct. However when the specification is checked with ZDRa, the message says that circular reasoning has been found and shows the path of the loop, which in this case is `CS4, TS4, TS5, TS6`. The error message

which appears specifically for this example in shown in figure A.4 in appendix A.4.3.

5.2.3 Products

When the specification has been ZDRa checked the program will then output two new files.

1. ZDRa specification Dependency Graph
2. ZDRa specification GoTo Graph

The ZDRa specification Dependency graph uses the labels and annotations from the ZDRa to show the dependencies between each of the instances. The ZDRa GoTo graph illustrates which instances are dependent or are needed for another instance to exist. Both graphs are built using the directed graph built in the ZDRa check (see section 5.2.2).

5.2.3.1 Dependency Graph

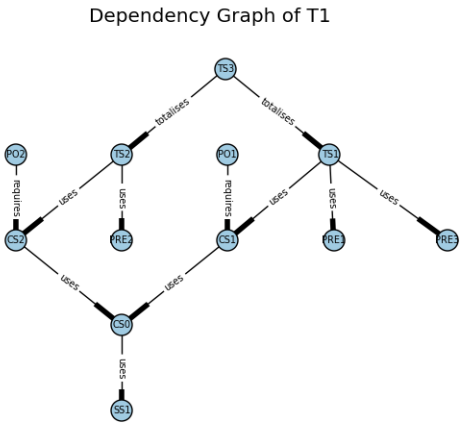


Figure 5.13: An example of a dependency graph.

An example of a *dependency graph* can be seen in figure 5.13. This image represents the compiled ZDRa annotated document but it graph form. All the boxes that show up in the compiled document are represented by nodes in the graph. The arrows

from the instances are represented by the edges in the graph, all the arrows in the document and edges in the graph should be pointing in the same direction.

5.2.3.2 GoTo Graph

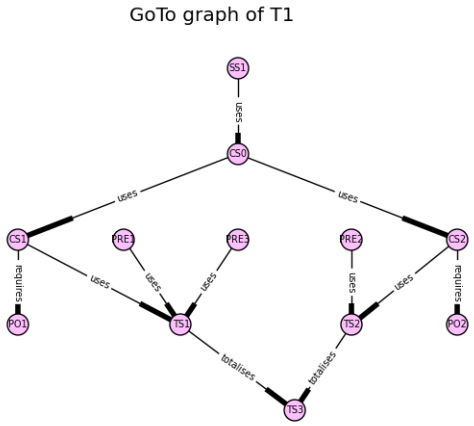


Figure 5.14: An example of a goto graph ZDRa correct.

An example of a *goto graph* is shown in figure 5.14, it is very similar to the *dependency graph*. The *goto graph* also uses the boxes created by the ZDRa annotations `draschema` and `draline`. However the only slight difference is the directions the edges are pointing to in some of the relations. For example if we had the relation `\initialOf{IS1}{SS1}`, in the compiled document and in the *dependency graph* the arrow will be going from `IS1` to `SS1` this is because it is indeed true that the initial schema `IS1` is the **initial of** the state schema `SS1`. On the other hand, in the *goto graph* the edge is pointing the other way from the stateschema `SS1` to the initialschema `IS1`. This is because the initialschema `IS1` needs `SS1` to exist, that is if `SS1` didn't exist then `IS1` couldn't initialise it. Therfore the instance `IS1` is dependent on `SS1`.

The other ZDRa relations which also reverse the direction of the arrow in the *goto graph* are *uses*,*requires* and *totalises*.

5.3 Conclusion

In this chapter the ZDRa step of the ZMathLang has been described. A LaTeX style package has been created to allow a user to annotate a Z specification and see the structure of the system. A ZDRa program has been created to check for rhetorical correctness and make sure there are no loops in the reasoning of the specification. Warning messages appear if the specification is still lacking some totalising schemas for some preconditions. If the specification is correct at the ZDRa stage then the user may then go on to create general and theorem prover specific skeletons which are described in the next chapter.

Chapter 6

From ZDRa to General Proof Sketch

The skeletons described in this chapter are automatically generated if the specification passes the ZDRa check. Section 6.1 describes the general proof skeleton. Which uses the graphs generated in the ZDRa to provide the order the instaces should go to into any theorem prover. Section 7 then explains how a general proof skeleton can be automatically translated into a skeleton in Isabelle format automatically. In section 7.4 we describe how the Isabelle Skeleton can be used to fully prove a formal specification which requires two steps, the first is an automatic step to fill in the Isabelle skeleton and the final step is up to the user to prove the lemma's and properties of the specification.

6.1 What is a General Proof Sketch

When checking for ZDRa correctness the program adds all the annotated chunks into a dependency graph and a GoTo graph. Both these graphs are directed graphs.

We then run an algorithm on the GoTo graph to generate a proof skeleton.

Algorithm 1 shows part of the code in generating this proof sketch.

```
Data: instances are known as 'nodes'
Data: OrderOfGraph is an ordered graph which lists the general proof
      skeleton
Data: ListOfAllNodes is an unordered list containing all nodes
for each node in ListOfAllNodes do
    if node is a root then
        push to OrderOfGraph
    else
        nothing
    end
end
for each node in ListOfAllNodes do
    if node in OrderOfGraph then
        remove node from ListOfAllNodes
    else
        nothing
    end
end
while there are nodes in ListOfAllNodes do
    for each node in ListOfAllNodes do
        predes = predecessor of each node;
        if predes is a subset of OrderOfGraph then
            push each node to OrderOfGraph;
            remove each node;
        else
            nothing
        end
    end
end
```

Algorithm 1: Part of the algorithm to create a proof sketch.

6.2 Creating the Graph

Here we show how a Proof skeleton is calculated using the Goto graph created when running the ZDRa check on a specification.

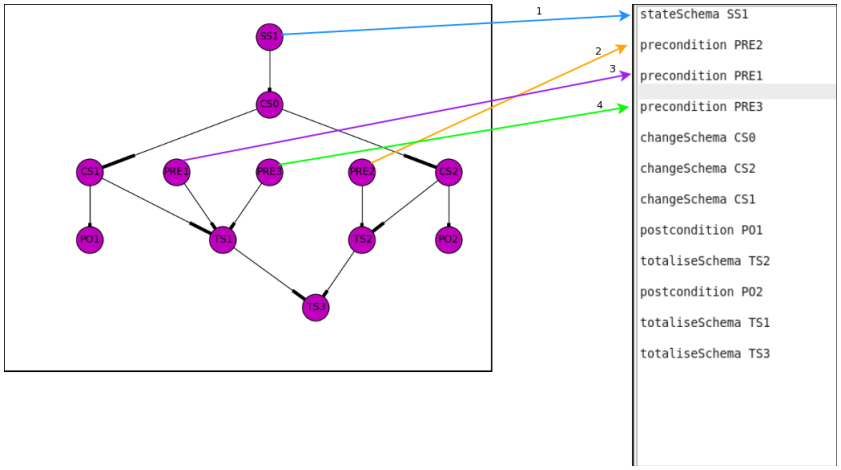


Figure 6.1: GoTo graph and proof skeleton of vending machine step 1.

First of all the program looks at all the nodes of the GoTo graph and prints out all the nodes which are not dependent on anything. That is, they may have successors but they have no predecessors, they do not use or need anything else and can stand by themselves. These nodes can be printed in any order, so in diagram 6.1 we see that we have SS1, PRE1 PRE2 and PRE3 all printed.

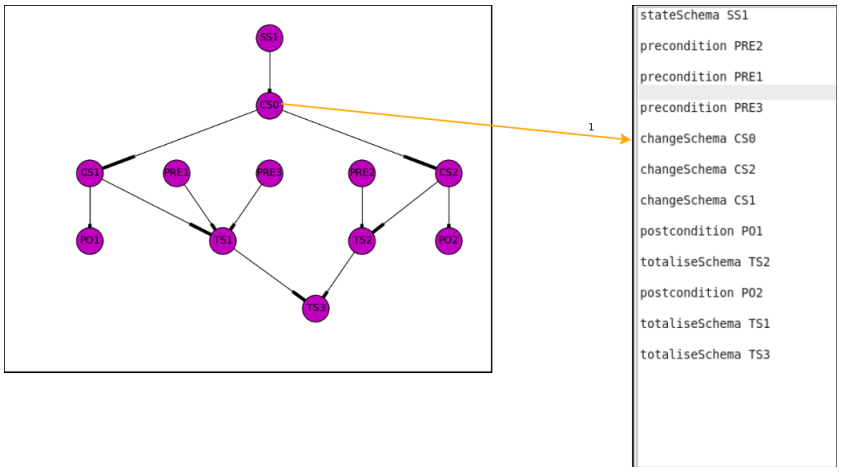


Figure 6.2: GoTo graph and proof skeleton of vending machine step 2.

The next part of the algorithm checks whether there exists a node in the GoTo graph where all of its parents are printed out in the proof skeleton. Figure 6.2 shows that the next node to be in the proof skeleton is CS0.

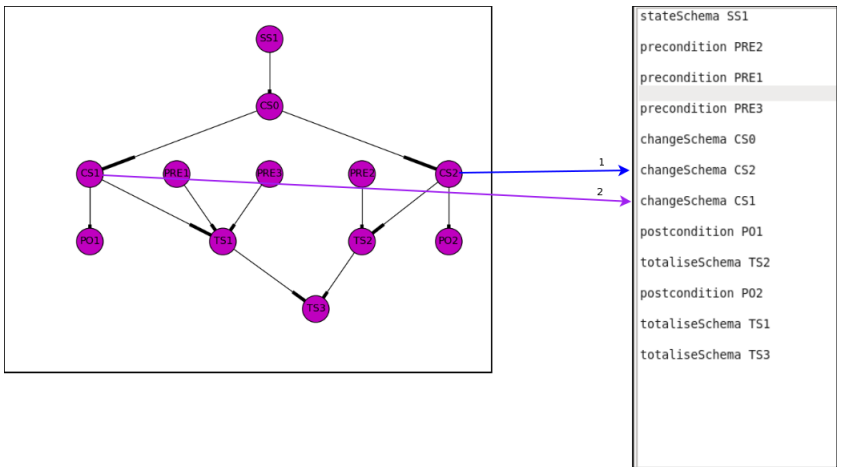


Figure 6.3: GoTo graph and proof skeleton of vending machine step 3.

The next part we see that after CS0 is added to the proof skeleton then both CS1, and CS2 can be added. This is shown in figure 6.3.

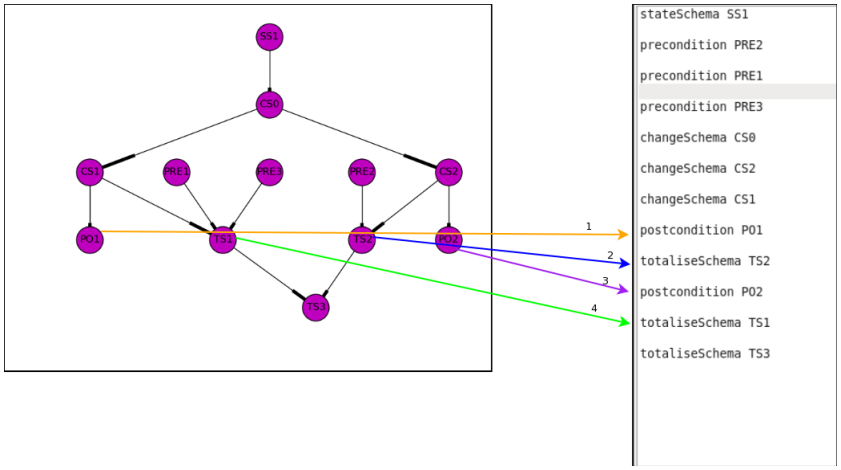


Figure 6.4: GoTo graph and proof skeleton of vending machine step 4.

Figure 6.4 shows the next stage of adding nodes to the Proof Skeleton. Since CS1 and CS2 are now added to the proof skeleton then the next row of nodes can be added. Since PO1 only had one parent (CS1) it is added first, PO2 also had one parent (CS2) it is added second. The others had more parents which are already in the proof sketch so they are added next randomly.

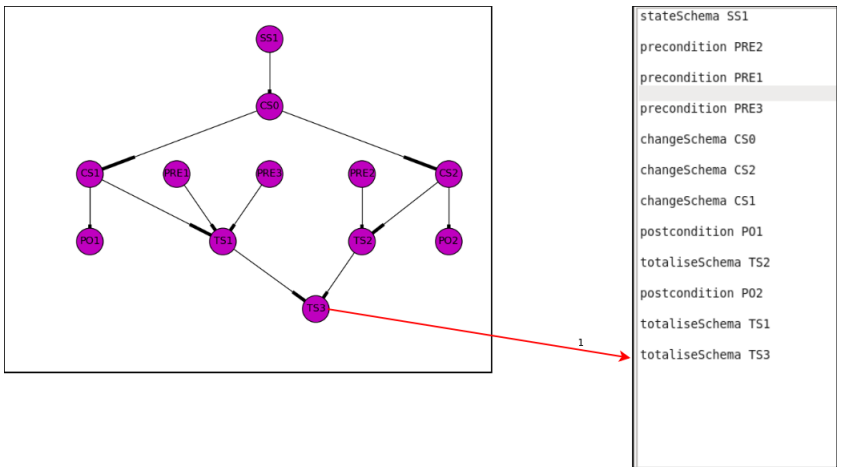


Figure 6.5: GoTo graph and proof skeleton of vending machine step 5.

We come to the final stage of the dependency graph, when all the nodes are in the dependency graph except for one which is added to the end.

6.3 Proof Obligations

There are many properties one may wish to prove about their specification. These certain properties are called proof obligations. Proof obligations for formal notations are an entire research area in their own right. However as the ZMathLang framework concentrates on giving the novice an idea of how to prove their specification we will focus on checking the specification for consistancy. Using the description in [78], checking the specification is consitant can fall under two categories:

- POb1, Feasability of an operation
- POb2, Other specific proof obligation for the chosen specification

We use the syntax *Context* $\vdash$ *predicate* taken from [78] to define the proof obligations.

6.3.1 POb1, Proof Obligation type 1

Definition 6.3.1. *POb1*

$$Context \vdash \exists(var :: type) * \bullet PRE\# \wedge PO\# \longrightarrow SI\# \wedge SI\#'$$

where $(var :: type)*$ *are the variables and types used in the schema with their types. We denote* $*$ *to declare there is 1 or many variables and types.* $SI\#$ *is the state Invariants of the specification,* $SI\#'$ *is the state invariants prime in the specification,* $PRE\#$ *is the precondition of the schema,* $PO\#$ *is the post operation of the schema and* $\#$ *is some arbitary number which the user has labeled their instance with.*

POb1 shows the feasability of an operation. When an operation can transfor a state to another state in the state space (a Δ schema). If an operation is feasible, the preconditional state and postconditional state should satisfy the state invariants of the specification.

Again by using the ModuleReg specification we can see a proof obligation is needed when adding a student doesnt change the state invariants of the specification.

```
lemma AddStudentDoesntChangeSI:
"(\<exists> taking taking'  :: (PERSON * MODULE) set.
 \<exists> degModules degModules':: MODULE set.
 \<exists> students students':: PERSON set.
  \<exists> p :: PERSON.
(students' = students \<union> {(p)})
\<and> (taking' = taking)
\<and> (degModules' = degModules)
\<longrightarrow> ((Domain taking \<subseteq> students)
\<and> (Range taking \<subseteq> degModules)
\<and> (Domain taking' \<subseteq> students')
\<and> (Range taking' \<subseteq> degModules'))) "
```

The ZDRa syntax of this proof obligation would be :

```
lemma AddStudentDoesntChangeSI:
" \<exists> (*CS1_variables :: CS1_TYPES*).
(PRE1)
```

```
\<and> (PO1)
\<longrightarrow> ((SI1)
\<and (SI1'))"
```

The ZDRa syntax of the proof obligation stays that there exists some variables of the operational schema where the precondition (PRE1) *and* postCondition (PO1) imply that the stateInvariants (SI1) and stateInvariants prime hold.

6.3.2 POb2, Proof Obligation type 2

As POb2 are any other relevent properties users wish to prove about the specification we can not formally define it. However an example would be if there existed a specification where an operator which added a member to a club and then removed a member from the club. Then the amount of members should be the same after both operators have completed the task.

One such example is in the ModuleReg specification the RegForModule schema postcondition shows that `(taking' = taking  {(p, m)})` therefore if this were to happen then we should make sure that `taking'` is not empty after the operation. This proof obligation is very specific to the ModuleReg specification and the user would need to write and check this themselves. To do such we have the following lemma:

```
lemma notEmpty:
"(taking' = taking \<union> {(p,m)})
\<longrightarrow> (taking' \<noteq> {})"
```

Where the name of the lemma is `notEmpty` then the postOperation of the ChangeSchema is `(taking' = taking \<union> {(p,m)})` then checking that the set is not empty follows the right arrow `(taking' \<noteq> {})`.

POb1 can be automate. Since POb2 is specification specific, each user will need to define these themselves if they so wish.

6.3.3 Proof Obligations in the General Proof Skeleton

Since the POb2 are specific to the specification only POb1 are automatically added. They are generated as '*lemma's*' in the general proof skeleton.

Data: listOfProofObligations is a set containing the ZDRa names of properties to prove
Data: i is the highest value of lemma, e.g if we have the following lemmas

$$L1, L2, L3$$

Data: setOfCS is a set containing all the ZDRa names which are a change schema
Data: lemmaset is a set containing all ZDRa names of existing lemmas

```
1  for (ZDRa name, instance type) in GoTo graph do
2  |  Add ZDRa name to lemmaSet
3  end
4  if there are elements in lemmaSet then
5  |  i becomes highest value lemma
6  else
7  |  i = 0
8  end
9  for (zdra Name, instance type) in GoTo graph do
10 |  if instance type is a changeSchema then
11 |  |  add zdra Name to setOfCS
12 |  else
13 |  |  nothing
14 |  end
15 end
16 if there are elements in setOfCS then
17 |  for (zdra name, instance type) in setOfCS do
18 |  |  indexOfElement = index of (zdra name, instance type);
19 |  |  numberOfNewLemma = i + indexOfElement + 1;
20 |  |  ZDRaNameOfNewLemma = 'L' + numberOfNewLemma;
21 |  |  push (ZDRaNameOfNewLemma, "lemma") to general proof skeleton;
22 |  end
23 else
24 |  nothing
25 end
```

Algorithm 2: Part of the algorithm to create Proof Obligation ZDRa names .

The proof obligations which check that changeSchema's and outputSchema's follow the stateInvariants are added to the original general proof skeleton. The general proof skeleton starts of with being an ordered list (GPSaOL), then the algorithm for generating a list of proof obligations is run and added to the original

proof skeleton. Algorithm 2 shows the algorithm which creates the ordered list of proof obligations.

Lines 338-344 find all the existing lemma's in the General Proof Skeleton ordered list (GpsaOL) and sets i to be the highest number. For example if there are existing lemmas in GpsaOL (L1, L2, L3), then i becomes 3. If there are no existing lemmas in GpsaOL then i stays as 0. Lines 347-250 take all the elements which are *changeSchema* instances and adds them to a list of *setCSandOS*. Then lines 350-359 loops through all the changeSchema's, takes the ZDRa name, add's L + a number + ZDRa name and adds it to the *listOfProofObligations*.

For Example if we had the following GpsaOL:
[(SS1, stateSchema), (IS1, initialSchema), (CS1, changeSchema), (CS2, changeSchema), (TS1, totaliseSchema)]
Then in this case:

- lemmaSet = []
- i = 0
- setofCsandOs = [(CS1, changeSchema), (CS2, changeSchema)]

Then for each element in setofCsandOs we would add the new elements (L1_CS1, lemma) and (L2_CS2, lemma) to the ordered list *listOfProofObligations*.

The new GpsaOL would then become [(SS1, stateSchema), (IS1, initialSchema), (CS1, changeSchema), (CS2, changeSchema), (TS1, totaliseSchema), (L1_CS1, lemma) and (L2_CS2, lemma)]

If for example the original GpsaOL was
[(SS1, stateSchema), (IS1, initialSchema), (CS1, changeSchema), (CS2, changeSchema), (TS1, totaliseSchema), (L1, lemma), (L2, lemma)]
Then then new proof obligation lemmas would be (L3_CS1, lemma) and (L4_CS2, lemma) as we would already have L1 and L2.

```
stateSchema SS1
initialSchema IS1
changeSchema CS1
changeSchema CS2
totaliseSchema TS1
lemma L1_CS1
lemma L2_CS2
```

Figure 6.6: Example of a General Proof Skeleton with lemma's.

An example of a General Proof Skeleton with added proof obligations is shown in Figure 6.6. The changeSchema's in this specification are CS1 and CS2. Therefore to make sure the changeSchemas do not change the state of the specification and comply with the state invariants the two lemma's L1_CS1 and L2_CS2 have been added.

6.3.3.1 Proof Obligations in specification examples

Since the vending machine specification (appendix A.1.5) doesn't have any stateInvaritants then the Gpsa will not have any added proof obligations to check for consistance. That is we can't check that the postconditions do not change the state if the state has no restrictions. However the birthdaybook example (appendix A.2.10) does have stateInvariants. Therfore we must add properties to check that any changeSchema's follow the state restrictions. Part of the birthdaybook Gpsa is shown in figure 6.7. Since there are stateInvariants (SI1) and a changing state Schema (CS1) then the proof obligation `L1_(CS1)` has been added to the Gpsa.

```
stateSchema SS1

stateInvarients SI1

initialSchema IS1

postcondition PO2

outputSchema OS1

precondition PRE2

changeSchema CS1

totaliseSchema TS1

.......

lemma L1_(CS1)
```

Figure 6.7: Part of the GPSa for the birthdayBook example. (Full version shown in appendix A.2.10)

6.4 Conclusion

This chapter describes how the ZDRa program uses the GoTo graph to generate a general proof skeleton (step 2→3 in figure 3.1). The general proof skeleton is an automatically generated .txt file which displays the order in which this instances must go in a theorem prover to be logically correct. This chapter gives a basic understanding of proof obligations for Z and examples which proof obligations are automatically generated when translating Z specifications into Isabelle. We give a formal definition of the proof obligation to check for consistancy of the state invariants and show an example. The next chapter describes how the general proof skeleton is translated into Isabelle syntax.

Chapter 7

General Proof Sketch aspect and beyond

When translating from the ZDRa annotated specification to the Isabelle skeleton, the syntax needed to be changed in order for the specification to parse through Isabelle. In this section we outline how the Z specification is translated into Isabelle.

If the user has labelled a theory in the specification (T#) then that will begin writing an Isabelle skeleton.

For example if we had an empty specification and named it a then the program will create an empty Isabelle skeleton such as:

```
theory gpsa_a
imports
main
begin
end
```

If the user labels a schema '`SS1`' meaning the stateSchema of the specification then that in Isabelle becomes a '`record`' and a '`locale`' is created. Using our example of specifcation named '`a`' we get the following (after the premuable descibed before):

```
record SS1 =
(*DECLARATIONS*)
```

```
locale a =
fixes (*GLOBAL DECLARATIONS*)
assumes SI#
begin
end
end
```

If there are no state invarients in the state schema at this point then there is no '`assumes SI#`') line.

All other schemas including changeSchemas, outputSchemas, preconditions that are schemas, post conditions that are schemas and all other state schema become definitions in the Isabelle skeleton. So for example if we have the following schema writte in ZDRa

```
\draschema{CS1}{
\begin{schema}{b}
someDeclaration
\where
\draline{PO1}{someExpression}
\end{schema}}
```

Then when translating into the Isabelle skeleton it becomes:

```
definition CS1 ::
"(*CS1_TYPES*) => bool"
where
"CS1 (*CS1_VARIABLES*) == PO1"
```

At this stage it doesn't matter what the declarations and expressions are as they get filled in at the next stage. The Isabelle skeleton only uses the ZDRa labels to be created.

Totalising schemas, written either horizontally or vertically in a specification become definitions when translating into the Isabelle skeleton. For example if we have the following totalisingSchema:

```
\draline{TS1}{someSchema == someExpression}
```

This would translate to the Isabelle skeleton as:

```
definition TS1 ::
"(*TS1_TYPES*) => bool"
where
"TS1 (*TS1_VARIABLES*) == (*TS1_EXPRESSION*)"
```

Again, at this stage it doesn't matter what the expression is. As it gets filled in at the next stage.

7.1 Proof Obligations in Isabelle Syntax

Lemmas which are proof obligations. That is instances with the a ZDRa name `L#_CS#` where '`#`' is a number become `lemma's` in Isabelle syntax. The translation from the General Proof Skeleton aspect (GPSa) into Isabelle syntax depends if the changeSchema in question has a precondition, postcondition or both. We use definition 6.3.1 in aid with the translation.

7.1.1 Proof Obligation translation where the schema has a precondition

If the changeSchema in which the proof obligation is about has a precondition as well as a postcondition then the translation will be as follows.

If an instance has the ZDRa name '`L1_CS1`' and we have the relations (CS1, requires, PRE1), (PRE1, allows, PO2) and (CS1, uses, IS1) then the Isabelle skeleton syntax would be as follows:

```
lemma L1_CS1:
" \<exists> (*CS1_variables :: CS1_TYPES*).
(PRE1)
\<and> (PO2)
\<longrightarrow> ((SI1)
```

```
\<and (SI1'))"
sorry
```

If the instance in the GPSa was 'L1_CS1' and the relationship only had a precondition and no post condition ie (CS1, requires, PRE1) and (CS1, uses, IS1) but not the allows relationship the syntax in the Isabelle skeleton would be

```
lemma L1_CS1:
" \<exists> (*CS1_variables :: CS1_TYPES*).
(PRE1)
\<longrightarrow> ((SI1)
\<and (SI1'))"
sorry
```

Where SI1 is the stateInvariants used in the stateSchema and SI1' is the stateInvariants prime.

7.1.2 Proof Obligation translation where the changeSchema has only postcondition

If the instance in the GPSa was L1_CS1 and CS1 only required a postcondition with no precondition that is had the relation (CS1, requires, PO2) and (CS1, uses, IS1) then the syntax in the Isabelle skeleton would be as follows:

```
lemma L1_CS1:
" \<exists> (*CS1_variables :: CS1_TYPES*).
(PO2)
\<longrightarrow> ((SI1)
\<and (SI1'))"
sorry
```

Where PO2 is the postcondition the changeSchema requires, SI1 is the stateInvariants in the stateSchema and SI1' is the stateInvariants prime.

We use the Isabelle word '`sorry`' to tell the theorem prover to skip a proof-in-progress and to treat the goal under consideration to be proved. This then causes the Isabelle skeleton to be an incorrect document but is a goal the user may prove at a later stage after the skeleton has been filled in.

7.2 Benefits

The Isabelle skeleton allows for incomplete specifications to also be automated into a half-baked proof. This way the user can have a general outline for their specification and fill in the missing information directly into the Isabelle skeleton at a later stage. For this reason it is good to have an Isabelle skeleton before the full filled in half-baked proof. The user then has an outline if they wish to add to the specification directly as they will have an example of how the instances should be translated. An example of a semi formal specification which has been translated to step 5 (the half baked proof) is shown in appendix A.5. Although this specification is not fully formal its grammatical correctness and rhetorical correctness can be checked. It can also start being translated into Isabelle syntax. The dependencies are shown between the current instances and the user can see what other information needs to be added for the specification to be complete.

7.3 ZCGa specification to Fill in the Isabelle Skeleton

Since translating using ZMathLang to translate Z specifications into an Isabelle skeleton can even been done on incomplete specifications, it is important to note that if some missing information is missing e.g. a declaration, expression etc then the comments of where these should go will not be changed. For example if we have the line `"(*CS1_TYPES*) => bool"` in the skeleton and the schema CS1 has no declarations yet then the line will not be changed, and it is up to the user to input the variables and the types of that definition.

It is important to note that all the ZCGa annotations at this stage disappear as the labelled information is automatically put into Isabelle syntax.

7.3.1 Z Types and FreeTypes

The program which fills in the Isabelle skeleton goes through the entire specification and adds any Z declared types and freetypes before the record. For example, if a specification has the following:

```
\begin{zed}
[STUDENT]
\end{zed}
```

Then the line `typedecl STUDENT` will be added after the first begin in the skeleton.

If the specification had the following freetype:

```
\begin{zed}
REPORT ::= ok | already\_known | not\_known
\end{zed}
```

Then again, in the same place as the Z-Types the line

```
datatype  REPORT = ok | already_known | not_known
```

is added to the skeleton.

7.3.2 Declarations

In Isabelle the types and variables are added speratly. For insance if we had the following schema:

```
\draschema{OS1}{
\begin{schema}{ab}
d: \power COLOUR
c: COLOUR
```

```
\where
\draline{PO1}{c \in d}
\end{schema}}
```

Then the Isabelle skeleton for this schema will be as follows:

```
definition OS1 ::
"(*OS1_TYPES*) => bool"
where
"OS1 (*OS1_VARIABLES*) == (PO1)"
```

Since we have two declarations, the filling in program would change the definition in the skeleton as follows:

```
definition ab ::
"COLOUR set => COLOUR => bool"
where
"ab d c == (c \<in> d)"
```

Therefore, from the declarations, the types replace the line `(*OS1_TYPES*)` and the variables replace the line `(*OS1_VARIABLES*)`.

7.3.3 Expressions

Since the majority of the syntax for expressions is very similar to the syntax in Isabelle, the expressions are put in directly with minor changes. The expressions replace the ZDRa labellings.

Using our previous example shown in the last section, we have the schema '`ab`', in the skeleton we have a label '`PO1`' which is then replaced by the expression `c \<in> d`. Note this expression is very similar to the expression in LaTeX `c \in d` apart from the symbol `\in` becomes `\<in>`. Table 7.1 shows the rest of these automatic changes of the syntax made from LaTeX to Isabelle.

Syntax in Z	Syntax in LaTeX	Syntax in Isabelle

{...}	\{...\}	{...}
(‖ ... ‖)	\limg...\rimg	\<lparr>...\<rparr>
⟨...⟩	\langle...\rangle	\<langle>...\<rangle>
# A	\#	card *if A is set*, length *if A is list*
∪	\cup	\<union>
∩	\cap	\<inter>
×	\cross	\<times>
\	\setminus	-
≥	\geq	\<ge>
≤	\leq	\<le>
◁	\lhd	\<lhd>
▷	\rhd	\<rhd>
⩤	\nrres	\<unlhd>
⩥	\ndres	\<unrhd>
⇒	\implies	\<Longrightarrow>
⇔	\iff	\<Longleftrightarrow>
∉	\notin	\<notin>
∈	\in	\<in>
⊂	\subset	\<subset>
⊆	\subseteq	\<subseteq>
∧	\land	\<and>
∨	\lor	\<or>
¬	\lnot	\<not>
≠	\neq	\<noteq>
$a \mapsto b$	a \mapsto b	(a,b) ‘ ’ *if set preceding using* \<rightharpoonup>
$\mathbb{P}\, A$	\power A	A set
$\mathbb{N}$	\nat	nat
$\mathbb{N}_1$	\nat_1	nat
$\mathbb{Z}$	\num	num
$A \nrightarrow B$	A \pfun B	(A \<rightharpoonup> B)
$A \rightarrow B$	A \fun B	(A * B) set
$A \leftrightarrow B$	A \rel B	(A * B) set
seq A	\seq A	A list
iseq A	\iseq A	A list
$\mathrm{seq}_1\, A$	\iseq_1 A	A list
dom A	\dom A	Domain A dom *if set preceding using* \<rightharpoonup>
ran A	\ran A	Range A ran *if set preceding using* \<rightharpoonup>
∃	\exists	\<exists>
∀	\forall	\<forall>
•	@	.
$R^{\sim}$	R\inv	\<R^-1>
R^k	R^{k}	\<R^k>

Table 7.1: A table showing the symbols which are changed from Z specifications in LaTeX to Isabelle.

Some symbols in the Z toolkit are the same regardless of whether they are a sequence/list/set or total function/partial function. However the syntax sometimes varies in Isabelle on these ocassions.

One part of the Z mathematical toolkit which we need to rewrite are the use of partial functions. In Isabelle/HOL all functions are total but there are ways to do certain proofs about partial functions [46]. We translate total functions as a set of pairs `(A * B) set`, we use the `\<rightharpoonup>` symbol to describe partial funcations. Any proofs to check for partial functions if needed can be done by the user in step 6 shown in figure 3.1 but the details of these proofs are beyond the scope of this thesis.

Other parts which differ are the following:

- '`\#`' is a symbol which takes a set or list [1] as a parameter and returns the number of elements within that set or list. In Z, '`\#`' can be used for both set and list, however in Isabelle we must translate '`\#`' into '`card`' if the parameter is a set or '`length`' if the parameter is a list.

- '`\mapsto`' is a symbol which takes two terms and returns a term. If we have a set '`funset`', which has type '`A \fun B`' and $f \mapsto s \in funset$ then in Isabelle the term '`f \mapsto s`' will be translated as '`(f,s)`'. However if '`funset`' had the type '`A \pfun B`' then the Isabelle translation of '`f \mapsto s`' would be '`f s`'.

- '`\dom`' and '`\ran`' in Z are the same regardless whether it is being applied to an element in a partial function or a total function. However in Isabelle the syntax varies between the two types of functions. If the '`dom`' being applied to an element '`k`', with type '`(A * B) set`' then the Isabelle translation would be '`Domain k`' whereas if '`k`' has type '`A \<rightharpoonup B`' then the '`domain`' of '`k`' in Isabelle would be translated as '`dom k`'. Similarly we would have '`Range k`' and '`ran k`' for total and partial functions of '`k`' respectively.

7.3.4 Schema Names

The Names of the Schema are added to the skeleton by using the ZDRa name. For example if the specification had the line `\draschema{TS1}{\begin{schema}{ab}{..`

[1] by list we mean an ordered set

then anywhere 'TS1' is listed in the skeleton it will be converted to 'ab'. This is done throughout the entire skeleton.

7.3.5 Proof Obligations

Using the birthdaybook specification an example we can see that we have the following schema:

```
\draschema{CS1}{
\begin{schema}{AddBirthday}
\text{\Delta BirthdayBook} \\
\text{\declaration{\term{name?}: \expression{NAME}}} \\
\text{\declaration{\term{date?}: \expression{DATE}}}
\where
\draline{PRE1}{\text{\expression{\term{name?} \notin \set{known}}}}\\
\draline{PO3}{\expression{\set{birthday'} = \set{\set{birthday} \cup \set{\{\te
\end{schema}}
\uses{CS1}{IS1}
\requires{CS1}{PRE1}
\allows{PRE1}{PO3}
```

The schema itself is represented in the filled in Isabelle syntax as:

```
definition AddBirthday ::
"(NAME set) \<Rightarrow> NAME \<Rightarrow> BirthdayBook \<Rightarrow> Birthda
where
"AddBirthday known' name birthdaybook birthdaybook' date birthday' ==
 (name \<notin> known)
\<and> (birthday' = birthday \<union> (name,date))"
```

Then the proof obligation which checks that the before state and after state of this changeSchema complies with the stateInvariants in represented as the following in the Isabelle Skeleton:

```
lemma CS1_L1:
"(\<exists> (*CS1_VARIABLESANDTYPES*).
(PRE1)
\<and> (PO3)
\<longrightarrow> ((SI1)
\<and (SI1'))""
sorry
```

This lemma filled in becomes the following proof obligation:

```
lemma AddBirthday_L1:
"(\<exists> known' :: (NAME set).
\<exists> name :: NAME.
\<exists> birthdaybook :: BirthdayBook.
\<exists> birthdaybook' :: BirthdayBook.
\<exists> date :: DATE.
\<exists> birthday' :: (NAME \<rightharpoonup> DATE).
(name \<notin> known)
\<and> (birthday' = birthday \<union> (name,date))
\<longrightarrow> ((known = dom birthday)
\<and> (known' = dom birthday')))"
```

7.4 Filled in Isabelle Skeleton to a Full Proof

The final step to get from a half-baked proof into a full proof is labelled as step 6 in Figure 3.1, this is also named fill in 2. Technically the specification the user automatically generates in fill in 1 is fully formalised in Isabelle if there are no other properties to be proved. If the specification is not fully formalised, using the half-baked proof generated in step 5, the user then adds any safety properties about the specification they wish to prove in the form of *lemmas*. As the properties will be specific to the user and/or specification it is difficult to automate this step. Therefore some theorem prover knowledge may be required for step 6. Some of

the automated theorem prover tools such as Sledgehammer [9] may be useful when proving the properties.

Figure 7.1 shows an example of a proof obligations generated by ZMathLang proved in Isabelle. Again we have highlighted the user input in red. To help prove these properties wed have used sledgehammer [9] which is part of the Isabelle/Hol package. The full proof of this specification can be found in appendix A.6. At this point, proving properties in a theorem prover may require some expertise in the field. Proving tools such as SMT solvers [27] such as sledgehammer are an interesting area of research on it's own however these such details are out with the scope of this thesis.

```
lemma AddStudent_L2:
"(\<exists> degModules:: MODULE set.
\<exists> students :: PERSON set.
\<exists> taking :: (PERSON * MODULE) set.
\<exists> p :: PERSON.
\<exists> degModules':: MODULE set.
\<exists> students' :: PERSON set.
\<exists> taking' :: (PERSON * MODULE) set.
((students' = students \<union> {(p)})
\<and> (degModules' = degModules)
\<and> (taking' = taking))
\<longrightarrow>
((Domain taking \<subseteq> students)
\<and> (Range taking \<subseteq> degModules)
\<and> (Domain taking' \<subseteq> students')
\<and> (Range taking' \<subseteq> degModules')))"
by blast
```

Figure 7.1: An example of a proof completed by user input.

7.5 Conclusion

This chapter described the final steps in computerising a formal specification into full proof. It demonstrates how the program uses the automatically generated general skeleton to create an Isabelle skeleton. From the Isabelle skeleton the user can then automatically fill in the skeleton using the ZCGa annoatated specification giving a halfbaked proof. The last step would be to fill in any mising proofs in the halfbaked proof, this is still a difficult step and may require some theorem prover knowledge however this part is difficult to automate as different system specifications have different properties users wish to prove, therefore tools such as sledgehammer [9] may be useful at this point.

In the next chapeter we demonstrate a full example of a specification being taken through all the steps of the ZMathLang framework.

Chapter 8

Formalising the ZDRa and Skeletons

Cite zenglars first year report on formalisations

In this chapter we take a formal view on the ZDRa and skeletons created. The ZCGa has already been semi formalised using weak type theory [40].

8.1 Formal View on ZDRa

We denote the star character '*' to denote one or many. We remind the reader that '$\mathbf{\Gamma}$' denotes the schematext of a specification, $\mathcal{Z}, \mathcal{E}, \mathcal{T}$ and $\mathbb{S}$ are correctly typed *declatation*, *expression*, *term* and *set* respectively (from table 4.1 in chapter 4).

Definition 8.1.1. *Let $\mathcal{E}$ be a correctly typed expression.*
Let $\mathcal{D}$ be a correctly typed definition.
Where $((\mathcal{Z} \bullet \mathcal{E}) \,^*\!| \, \mathcal{T} \,^*\!| \, \mathbb{S}^*) \in \mathcal{E}$ and $(\mathcal{T} \,^*\!| \, \mathbb{S}^*) \in \mathcal{D}$
Let $\mathcal{ED}$ be the set containing $\mathcal{E}$'s and $\mathcal{D}$'s.

For example an expression can be the following: `t = t` where t is a term. Therefore this expression contains two terms. Another example of an expression could be: $\forall\, t : S \bullet t = 0$. This expression contains a declaration with an expression (a $(\mathcal{Z} \bullet \mathcal{E})$, which is $t : S$.

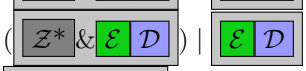
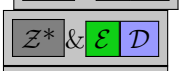
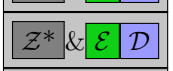
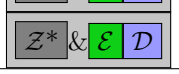
Instance	Allowed weak types	Annotations
precondition	$\mathcal{ED} \in \mathbf{\Gamma}$	
postcondition	$\mathcal{ED} \in \mathbf{\Gamma}$	
output	$\mathcal{ED} \in \mathbf{\Gamma}$	
stateInvariants	$\mathcal{ED} \in \mathbf{\Gamma}$	
stateSchema	$(\mathcal{Z}\&\mathcal{ED}) \mid \mathcal{Z} \in \mathbf{\Gamma}$	
theory	$(\mathcal{Z}\&\mathcal{ED}) \mid \mathcal{Z} \in \mathbf{\Gamma}$	
changeSchema	$(\mathcal{Z}\&\mathcal{ED}) \mid \mathcal{ED} \in \mathbf{\Gamma}$	
totaliseSchema	$(\mathcal{Z}\&\mathcal{ED}) \mid \mathcal{ED} \in \mathbf{\Gamma}$	
axiom	$\mathcal{Z}\&\mathcal{ED} \in \mathbf{\Gamma}$	
outputSchema	$\mathcal{Z}\&\mathcal{ED} \in \mathbf{\Gamma}$	
initSchema	$\mathcal{Z}\&\mathcal{ED} \in \mathbf{\Gamma}$	

Table 8.1: ZCGa annotations allowed in ZDRa instances

Table 8.1 shows which ZCGa are allowed to be in certain ZDRa instances. Where '&' means these weak type categories must be included in the instance. For example a *stateSchema* can be made up of one or more $\mathcal{Z}$ **AND** one or more $\mathcal{ED}$ **OR** one or more $\mathcal{Z}$ on their own. Therefore a *stateSchema* can not be made up of $\mathcal{ED}$ on it's own.

precondition, postcondition, output, stateInvariants. These instances can only contain a correctly typed expression or definition within the schematext. There may be one or more expressions or definitions.

stateSchema, theory. These instances can only contain one or more correctly typed declaration *and* one or more correctly typed expression and definition or one or more correctly typed declarations.

changeSchema, totalise. These instances can only contain one or more correctly typed declaration *and* one or more correctly typed expression and definition or one or more correctly typed expressions or definitions.

axiom, outputSchema, initSchema These instances can only contain one or more correctly typed declarations *and* one or more correctly typed expression and

definitions.

8.1.1 Properties

Since we have formally outlined the ZDRa in table 8.1 we can now write some properties about the ZDRa instances and relations.

Theorem 8.1.1. *Using table 8.1, the initialOf relation only permits the annotations:* $\mathcal{Z}\&\mathcal{ED}$, *initialOf*, $(\mathcal{Z}\&\mathcal{ED}) \mid \mathcal{Z}$.

Proof. From table 5.3 (section 5.1.2) we can see the only legal relation for initialOf is 'initSchema $\longrightarrow$ stateSchema'. An initSchema can only have the annotation $\mathcal{Z}\&\mathcal{ED}$ and a stateSchema can only have the annotations $(\mathcal{Z}\&\mathcal{ED}) \mid \mathcal{Z}$ (from table 8.1). Since $\longrightarrow$ represents the relation *initialOf* then we get $\mathcal{Z}\&\mathcal{ED}$, *initialOf*, $((\mathcal{Z}\&\mathcal{ED}) \mid \mathcal{Z})$. □

Theorem 8.1.2. *Using table 8.1, the allows relation only permits the annotations:* $\mathcal{ED}$, *allows*, $\mathcal{ED}$.

Proof. The allows relation only permits 'precondition $\longrightarrow$ postcondition' (table 5.3). Both a precondition and postcondition have the ZCGa annotations $\mathcal{ED}$. Since $\longrightarrow$ represents the relation. Then the result is $\mathcal{ED}$, *allows*, $\mathcal{ED}$. □

Theorem 8.1.3. *Using table 8.1, the requires relation only permits the annotations:*

- $\mathcal{Z}\&\mathcal{ED}$, *requires*, $\mathcal{ED}$
- $(\mathcal{Z}\&\mathcal{ED}) \mid \mathcal{ED}$, *requires*, $\mathcal{ED}$

Proof. Using table (table 5.3) of permitted relations the requires relation has the following syntax:

1. 'outputSchema $\longrightarrow$ precondition'
2. 'outputSchema $\longrightarrow$ output',
3. 'changeSchema $\longrightarrow$ precondition',
4. 'changeSchema $\longrightarrow$ postcondition',

- Using the allowed ZCGa annotations in table 8.1 we can see that an outputSchema contains only $\mathcal{Z}\&\mathcal{ED}$ and a precondition and output both only contain $\mathcal{ED}$ thus 1 and 2 become $\mathcal{Z}\&\mathcal{ED} \longrightarrow \mathcal{ED}$. Since $\longrightarrow$ represents the relation then we end up with $\mathcal{Z}\&\mathcal{ED}, \mathit{requires}, \mathcal{ED}$.

- The allowed ZCGa annotations for changeSchema are $(\mathcal{Z}\&\mathcal{ED}) \mid \mathcal{ED}$ and the allowed ZCGa annotations for both precondition and postcondition are $\mathcal{ED}$ therefore we end up with $((\mathcal{Z}\&\mathcal{ED}) \mid \mathcal{ED} \longrightarrow \mathcal{ED}$. Again the $\longrightarrow$ represents the relation so the $\longrightarrow$ becomes 'requires' and we are left with $(\mathcal{Z}\&\mathcal{ED}) \mid \mathcal{ED}, \mathit{requires}, \mathcal{ED}$.

□

Theorem 8.1.4. *Using table 8.1, the totalises relation only permits the annotations:*

- $(\mathcal{Z}\&\mathcal{ED}) \mid \mathcal{ED}), \mathit{totalises}, (\mathcal{Z}\&\mathcal{ED}) \mid \mathcal{ED}$
- $(\mathcal{Z}\&\mathcal{ED}) \mid \mathcal{ED}), \mathit{totalises}, \mathcal{Z}\&\mathcal{ED}$

Proof. Using table (table 5.3) of permitted relations the totalises relation has the following syntax:

1. 'totaliseSchema $\longrightarrow$ changeSchema'
2. 'totaliseSchema $\longrightarrow$ outputSchema'
3. 'totaliseSchema $\longrightarrow$ totaliseSchema'

- Table 5.3 shows the ZCGa syntax for totaliseSchema and changeSchema is $(\mathcal{Z}\&\mathcal{ED}) \mid \mathcal{ED})$. Thus the permitted relations would be $(\mathcal{Z}\&\mathcal{ED}) \mid \mathcal{ED}) \longrightarrow (\mathcal{Z}\&\mathcal{ED}) \mid \mathcal{ED})$ for points 1 and 3. Since $\longrightarrow$ represents the relation 'totalises' in this case we would have $(\mathcal{Z}\&\mathcal{ED}) \mid \mathcal{ED}), \mathit{totalises}, (\mathcal{Z}\&\mathcal{ED}) \mid \mathcal{ED})$. Hence the first part of the theorem is proven.

- Output schema has the ZCGa sytntax $(\mathcal{Z}\&\mathcal{ED})$ according to table 8.1. If totaliseSchema has the ZCGa syntax $(\mathcal{Z}\&\mathcal{ED}) \mid \mathcal{ED})$ then using point 2 the permitted relation would be $(\mathcal{Z}\&\mathcal{ED}) \mid \mathcal{ED}) \longrightarrow \mathcal{Z}\&\mathcal{ED}$. Again since $\longrightarrow$ represent totalise in this case we conclude with $(\mathcal{Z}\&\mathcal{ED}) \mid \mathcal{ED}), \mathit{totalises}, \mathcal{Z}\&\mathcal{ED}$.

□

Theorem 8.1.5. *Using table 8.1, the uses relation only permits the annotations:*

- $\mathcal{Z}\&\mathcal{ED}, uses, (\mathcal{Z}\&\mathcal{ED}) \mid \mathcal{Z}$
- $(\mathcal{Z}\&\mathcal{ED}) \mid \mathcal{ED}, uses, (\mathcal{Z}\&\mathcal{ED}) \mid \mathcal{Z}$
- $(\mathcal{Z}\&\mathcal{ED}) \mid \mathcal{Z}, uses, (\mathcal{Z}\&\mathcal{ED}) \mid \mathcal{Z}$
- $(\mathcal{Z}\&\mathcal{ED}) \mid \mathcal{Z}, uses, \mathcal{Z}\&\mathcal{ED}$
- $\mathcal{Z}\&\mathcal{ED}, uses, \mathcal{Z}\&\mathcal{ED}$
- $(\mathcal{Z}\&\mathcal{ED}) \mid \mathcal{ED}, uses, \mathcal{Z}\&\mathcal{ED}$

Proof. Using table (table 5.3) of permitted relations the uses relation has the following syntax:

1. 'outputSchema $\longrightarrow$ stateSchema'
2. 'changeSchema $\longrightarrow$ stateSchema'
3. 'stateSchema $\longrightarrow$ stateSchema'
4. 'stateSchema $\longrightarrow$ axiom'
5. 'outputSchema $\longrightarrow$ axiom'
6. 'changeSchema $\longrightarrow$ axiom'

- According to table 8.1 the ZCGa syntax for outputSchema is $\mathcal{Z}\&\mathcal{ED}$ and the syntax for stateSchema is $(\mathcal{Z}\&\mathcal{ED}) \mid \mathcal{Z}$. Therefore using point 1 we get '$\mathcal{Z}\&\mathcal{ED} \longrightarrow (\mathcal{Z}\&\mathcal{ED}) \mid \mathcal{Z}$'. Since $\longrightarrow$ represents the relation uses in this case we get $\mathcal{Z}\&\mathcal{ED}, uses, (\mathcal{Z}\&\mathcal{ED}) \mid \mathcal{Z}$.
- The syntax for changeSchema is $(\mathcal{Z}\&\mathcal{ED}) \mid \mathcal{ED}$ and the syntax for stateSchema is $(\mathcal{Z}\&\mathcal{ED}) \mid \mathcal{Z}$ according from table 8.1. Substituting the ZCGa for instance names in point 2 we get $(\mathcal{Z}\&\mathcal{ED}) \mid \mathcal{ED} \longrightarrow (\mathcal{Z}\&\mathcal{ED}) \mid \mathcal{Z}$. By substituting the relation uses for $\longrightarrow$ we end up with $(\mathcal{Z}\&\mathcal{ED}) \mid \mathcal{ED}, uses, (\mathcal{Z}\&\mathcal{ED}) \mid \mathcal{Z}$.

- Using table 8.1 the ZCGa syntax for stateSchema is $(\mathcal{Z}\&\mathcal{ED}) \mid \mathcal{Z}$. Using point 3, we substitute the ZCGa syntax for the instance name and we get $(\mathcal{Z}\&\mathcal{ED}) \mid \mathcal{Z} \longrightarrow (\mathcal{Z}\&\mathcal{ED}) \mid \mathcal{Z}$. Since $\longrightarrow$ represents uses then we get $(\mathcal{Z}\&\mathcal{ED}) \mid \mathcal{Z}, \mathit{uses}, (\mathcal{Z}\&\mathcal{ED}) \mid \mathcal{Z}$.

- The ZCGa syntax for stateSchema is $(\mathcal{Z}\&\mathcal{ED}) \mid \mathcal{Z}$ and the ZCGa syntax for axiom is $\mathcal{Z}\&\mathcal{ED}$. By subsituting these ZCGa syntax's in point 4 we get $(\mathcal{Z}\&\mathcal{ED}) \mid \mathcal{Z} \longrightarrow \mathcal{Z}\&\mathcal{ED}$. Since $\longrightarrow$ in this case is the relationship uses, we can change $\longrightarrow$ to 'uses' and get $(\mathcal{Z}\&\mathcal{ED}) \mid \mathcal{Z}, \mathit{uses}, \mathcal{Z}\&\mathcal{ED}$.

- According to table 8.1 the ZCGa syntax for outputSchema and axiom are both $\mathcal{Z}\&\mathcal{ED}$. Therefore if we put in the ZCGa syntax instead of the instance names in point 5 we get $\mathcal{Z}\&\mathcal{ED} \longrightarrow \mathcal{Z}\&\mathcal{ED}$. From table 5.3 we can deduce that $\longrightarrow$ mean 'uses' so we finish with $\mathcal{Z}\&\mathcal{ED}, \mathit{uses}, \mathcal{Z}\&\mathcal{ED}$.

- Finally, the ZCGa syntax for changeSchema is $(\mathcal{Z}\&\mathcal{ED}) \mid \mathcal{ED}$ and the ZCGa syntax for axiom is $\mathcal{Z}\&\mathcal{ED}$ according to table 8.1. If we substitute the allowed ZCGa syntax in point 6 we get $(\mathcal{Z}\&\mathcal{ED}) \mid \mathcal{ED} \longrightarrow \mathcal{Z}\&\mathcal{ED}$. Since $\longrightarrow$ is the relation 'uses' in this case we end up with $(\mathcal{Z}\&\mathcal{ED}) \mid \mathcal{ED}, \mathit{uses}, \mathcal{Z}\&\mathcal{ED}$.

□

8.1.2 Conclusion

In this section we have described the allowed syntax once the specification has been annotated with both ZCGa and ZDRa. We have used this syntax along with the relations permitted from table 5.3 in section 5.1.2 we have identified theorems and proved them about the ZMathLang syntax. The next step is to describe the syntax of the dependency graph. Which shows the instances and relations but without the content of the specification.

8.2 Formal View on Dependency Graph

Definition 8.2.1 (Nodes)**.** *We can say a node of the graph (or vertex)* $\mathbb{V}$*, is a pair.*

(id, ins)

of a unique identifier id, and instance name ins.

Definition 8.2.2 (Relational Edges)**.** *We can say a relation edge* $\mathbb{E}$*, is a triple*

$$(n_1,\ rel,\ n_2)$$

of a relation from node v_1 *to* v_2 *with relation name rel.*

We can now show the how annotations are represented and processed for the dependency graph.

Definition 8.2.3 (Dep graph set)**.** *A dependency graph* $\mathbb{D}$*, is a set of relational edges and a set of nodes.*

$\mathbb{D} = \mathbb{E} \cup \mathbb{V}$

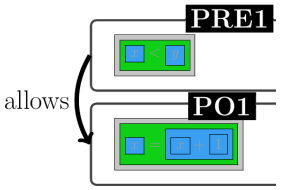
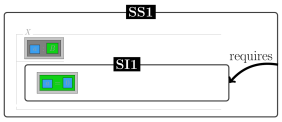
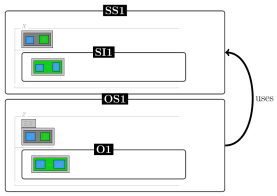
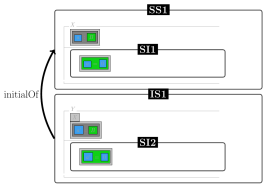
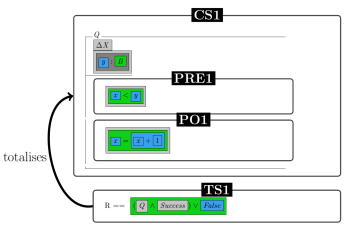
Nº	Annotations	
dra1)		$\mathbb{V}_{dra1} =$ $\{(PRE1, precondition), (PO1, postcondition)\}$ $\mathbb{E}_{dra1} =$ $\{((PRE1, precondition), allows, (PO1, postcondition))\}$ $\mathbb{D}_{dra1} = \mathbb{V}_{dra1} \cup \mathbb{E}_{dra1}$
dra2)		$\mathbb{V}_{dra2} =$ $\{(SS1, stateSchema), (SI1, stateInvariants)\}$ $\mathbb{E}_{dra2} =$ $\{((SS1, stateSchema), requires, (SI1, stateInvariants))\}$ $\mathbb{D}_{dra2} = \mathbb{V}_{dra2} \cup \mathbb{E}_{dra2}$
dra3)		$\mathbb{V}_{dra3} =$ $\{(SS1, stateSchema), (SI1, stateInvariants)$ $(OS1, outputSchema), (O1, output)\}$ $\mathbb{E}_{dra3} =$ $\{((OS1, outputSchema), uses, (SS1, stateSchema))\}$ $\mathbb{D}_{dra3} = \mathbb{V}_{dra3} \cup \mathbb{E}_{dra3}$
dra4)		$\mathbb{V}_{dra4} =$ $\{(SS1, stateSchema), (SI1, stateInvariants)$ $(IS1, initSchema), (IS2, stateInvariants)\}$ $\mathbb{E}_{dra4} =$ $\{((IS1, initSchema), initialOf, (SS1, stateSchema))\}$ $\mathbb{D}_{dra4} = \mathbb{V}_{dra4} \cup \mathbb{E}_{dra4}$
dra5)		$\mathbb{V}_{dra5} =$ $\{(CS1, changeSchema), (PRE1, precondition)$ $\{(PO1, postcondition), (TS1, totaliseSchema)\}$ $\mathbb{E}_{dra5} =$ $\{((TS1, totaliseSchema), totalises, (CS1, changeSchema))\}$ $\mathbb{D}_{dra5} = \mathbb{V}_{dra5} \cup \mathbb{E}_{dra5}$

Table 8.2: Using the formalised definitions for vertices and edges to create a dependency graph.

Table 8.2 show how a dependency graph can be created using the formal definitions for vertices and edges. The dependency graph is directly generated from the ZDRa annotated document. The relations *intialOf* and *uses* are used between schemas, *requires* becomes a childOf the schema which requires it and *allows* can be used between instances within the schema (see table 5.3). If the *allows* relationship is used within the schema, then both the precondition and postcondition/output becomes childrenOf the schema. Figure 8.1 shows two separate schemas which are not nested within eachother, however figure 8.2 shows the precondition PRE2 allows the postcondition PO2 within the schema CS1, since CS1 requires PRE2 then both PRE2 and PO2 are chidrenOf CS1.

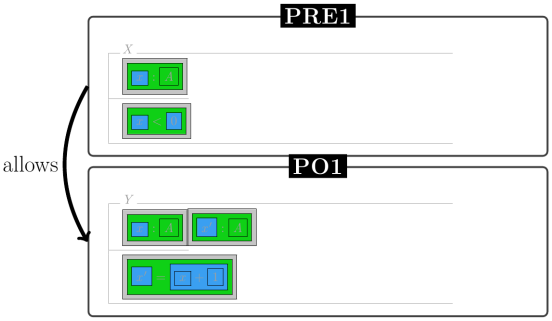


Figure 8.1: Relation with un-nested precondition and postcondition.

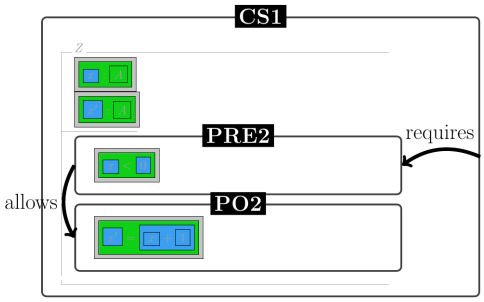


Figure 8.2: Relation with nested precondition and postcondition.

If we combine all the annotations in table 8.2 (whilst adding PRE1 and PO1 in a schema named CS1) we get a fully annotated specification and its dependency graph shown in figure 8.3 and 8.4 respectively.

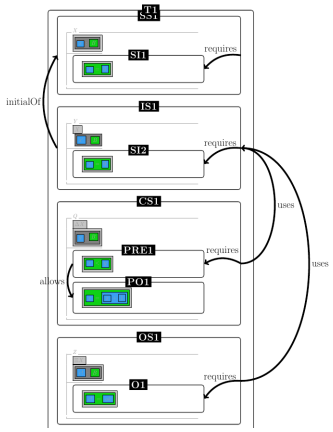


Figure 8.3: All annotations from table 8.2 combined into one specification.

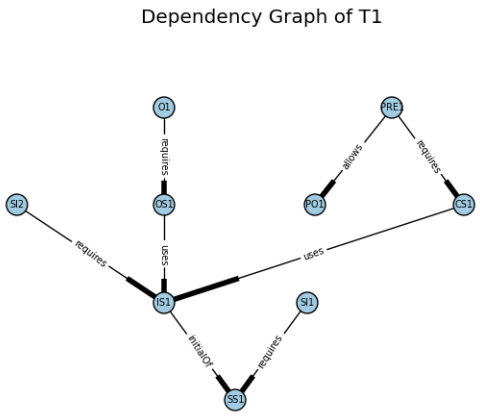


Figure 8.4: Dependency graph of the example in 8.3

Since figure 8.3 is a combination of the annotations in table 8.2 we can call this dependency graph $\mathbb{D}_{comb}$ where

$\mathbb{V}_{comb} = \{$

$(T1, theory), (SS1, stateSchema), (SI1, stateInvariants),$

$(IS1, initSchema), (SI2, stateInvariants), (CS1, changeSchema),$

$(PRE1, precondition), (PO1, postcondition), (OS1, outputSchema), (O1, output)\}$

and $\mathbb{E}_{comb} = \{$

$((SS1, stateSchema), requires, (SI1, stateInvariants))$

$((IS1, initSchema), requires, (SI2, stateInvariants))$

$((IS1, initSchema), initialOf, (SS1, stateSchema))$

$((CS1, changeSchema), requires, (PRE1, precondition))$

$((PRE1, precondition), allows, (PO1, postcondition))$

$((CS1, changeSchema), uses, (IS1, initSchema))$

$((OS1, outputSchema), requires, (O1, output))$

$((OS1, outputSchema), uses, (SS1, stateSchema))$

$\}$

and $\mathbb{D}_{comb} = \mathbb{V}_{comb} \cup \mathbb{E}_{comb}$

8.2.1 Conclusion

In this section we have looked at the dependency graph and defined it formally using sets. We have defined the nodes of a graph as a pair and the edges as a triple. We have given examples of certain annotations and how they would be represented formally. Although the dependency graph helps to see the relationships between instances we then need to order them correctly in order to input the specification into a theorem prover. To do this we need to determine the *textual order* of the relations. The textual order of relations is described in the next section.

8.3 Generation of the GoTo graph

Since the dependency graph only follows the annotations of the ZDRa we need to highlight the textual order of these annotations. For example some annotations may be stronger then others. The order in which instances are inputted in the theorem prover are also important so that the theorem prover may be parsed correctly. The GoTo graph is generated using the textual order of the dependency graph.

Definition 8.3.1 (Textual order)**.** *We can now formalise three different kinds of textual order when translating a dependency graph into the GoTo graph.*

- ***Strong textual order*** $\prec$***:*** *This order describes an entire entity relation between two nodes (id_A uses id_B would be written as $id_B \prec id_A$, id_A initialOf id_B would be written as $id_B \prec id_A$, id_A totalises id_B would be written as $id_B \prec id_A$).*

- ***Weak textual order*** $\preceq$***:*** *This order describes a sub-part relation between two nodes (id_A allows id_B within a schema written as $id_A \preceq id_B$).*

- ***Common textual order*** $\leftrightarrow$***:*** *This order describes a relation between two nodes where they are both dependent on each other (Where a draschema id_A requires a draline of some sort id_B is written as $id_A \leftrightarrow id_B$).*

The dependency graph is a direct representation of the ZDRa annotations and arrows represented when compiling the ZDRa annotated document. The GoTo

uses these annotations and describes the textual order of them. When inputting a specification into an automatic theorem prover, the textual order is important as it decides which part of the specification needs to be in-putted first in order to parse correctly.

Relation	**Meaning**	**Order**
id_A, initialOf, id_B	id_A is the initial schema of id_B or id_A initialises id_B	$id_B \prec id_A$
id_A, uses, id_B	id_B is being used in id_A or id_B is needed in id_A	$id_B \prec id_A$
id_A, totalises, id_B	id_A makes the precondition in id_B total	$id_B \prec id_A$
id_A, allows, id_B	id_A is allowing id_B to occus	$id_A \preceq id_B$
id_A, requires, id_B	id_A is requiring id_B to function	$id_A \leftrightarrow id_B$

Table 8.3: Example of ZDRa annotations and the textual order of them.

Definition 8.3.2. *We denote v_1 and v_2 to describe node 1 and node 2 respectively for example if we have* $\backslash initialof\{id_1\}\{id_2\}$ *and* id_1 *is IS1 and* id_2 *is SS1 then v_1 would be (IS1, initialSchema) and v_2 would be (SS1, stateSchema).*

```
1  goto_graph = directedGraph ;
2  dependency_graph = directedGraph ;
3  if \initialof{id_1}{id_2} then
4  |  addEdge(v_1, →, v_2) to dependency_graph ;
5  |  addEdge(v_2, →, v_1) to goto_graph;
6  if \uses{id_1}{id_2} then
7  |  addEdge(v_1, →, v_2) to dependency_graph ;
8  |  addEdge(v_2, →, v_1) to goto_graph ;
9  if \allows{id_1}{id_2} then
10 |  addEdge(v_1, →, v_2) to dependency_graph ;
11 |  addEdge(v_1, →, v_2) to goto_graph ;
12 if \requires{id_1}{id_2} then
13 |  addEdge(v_1, →, v_2) to dependency_graph ;
14 |  addEdge(v_1, →, v_2) to goto_graph ;
15 if \totalises{id_1}{id_2} then
16 |  addEdge(v_1, →, v_2) to dependency_graph ;
17 |  addEdge(v_2, →, v_1) to goto_graph ;
```

Algorithm 3: Algorithm to generate the dependency graph and goto.

Algorithm 3 shows the pseudocode of the implementation on how the dependency graph and goto graphs are created. It reads the labels created by the user when annotating the formal specification.

Note the edges for the relations *allows* and *requires* have the same direction both in the dependency graph and the goto graph. This is because if instance v_1 *allows* another instance v_2 to happen then instance v_1 must exist for instance v_2 to exist. The relationship *requires* is between a drashchema and a draline where a draschema requires a particular draline, in the algorithm we have the dependency graph edge and goto graph edge point in the same direction.

With the edges for relations *initialof*, *uses* and *totalises*, algorithm 3 changes the direction of the edges from the dependency graph to the goto graph. If a node v_1 is the initialOf another node v_2 then v_1 initialises v_2, therefore v_2 must exist

first for v_1 to initialise it. If a node v_1 uses another node v_2 then v_2 must exist first before it can be used by v_1. This is the same with the relation *totalises*, if a node v_1 totalises another node v_2 then the node v_2 must exist before the node v_1 can totalise it.

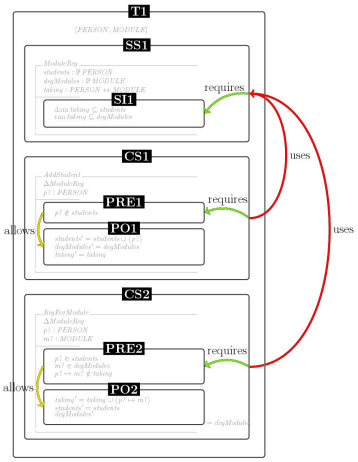


Figure 8.5: User annotated in ZDRa for the modulereg specification with arrows coloured.

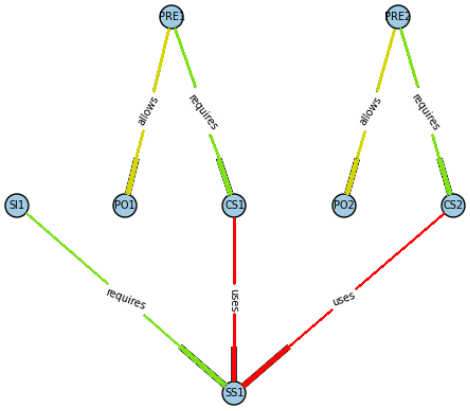


Figure 8.6: The dependency graph of modulereg specification with arrows coloured.

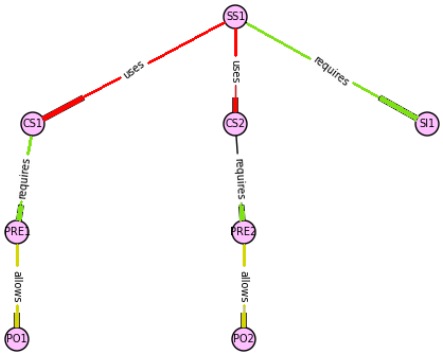


Figure 8.7: The goto graph of modulereg specification with arrows coloured.

Figure 8.5 shows the moduleReg specification annotated in ZDRa however the colours of the arrows have been changed as to compare it with the automatically generated dependency graph (figure 8.6) and it's corresponding goto grah (figure 8.7). Note the red arrows which correspond to the 'uses' relation are facing the same direction in the user annotated document and in the dependency graph (from

CS1 to SS1 and from CS2 to SS1) however they go the opposite direction in the goto graph. This goes the same for the green arrows which represent the 'requires' relation. On the other hand the yellow arrow, representing the 'allows' relation goes in the same direction in all three figures.

8.3.1 Conclusion

In this section we have highlighted why the goto grph is important and divded the ZDRa annotations into their correctly textual ordered categories (strong, weak and common). We added an algorithm to this section to describe how the dependency graph and goto graph are created from the ZDRa annotations inputted by the user. We also compared the ZDRa annotated specification and its corresponding dependency and goto graphs in more detail and shown which arrows change direction in practice. In the next section we take a formal look at the the goto graph.

8.4 Formal View on the GoTo Graph

Definition 8.4.1 (Textual order edge)**.** *We can say a textual order edge* $\mathbb{O}$ *is a quadruple*

$$(v_A, v_B, rel, to)$$

where it is a relation rel, from node v_A *to node* v_B *and it's textual order to where* $to \in \{\prec, \preceq, \leftrightarrow, \succ, \succeq\}$.

Definition 8.4.2 (Graph of Textual order)**.** *A graph of textual order is a pair (V,O), made up of a set of vertices V and a set of ordered edges O.*

The main reason for producing a GoTo graph is to automatically produce a skeleton for a certain theorem prover.

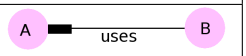
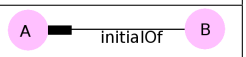
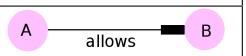
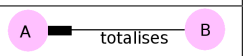
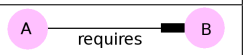
Relation	Textual Order	GoTo Edge
$(v_A,$ uses, $v_B)$	$v_A \succ v_B$	
$(v_A,$ initialof, $v_B)$	$v_A \succ v_B$	
$(v_A,$ allows, $v_B)$	$v_A \preceq v_B$	
$(v_A,$ totalises, $v_B)$	$v_A \succ v_B$	
$(v_A,$ requires, $v_B)$	$v_A \leftrightarrow v_B$	

Table 8.4: The relations representedby textual order and in the goto graph

Table 8.4 shows what the textual order reprsentation of each relation is and how it is represented as an edge in the GoTo graph. The uses relation has the texutal order '$\succ$' where the goto graph edge would be $(v_A, v_B, uses, \succ)$ as the relation uses would have a strong textual order. the initialof relation is similar to the uses relation where the syntax in the goto graph would be $(v_A, v_B, initialof, \succ)$. Totalises relation also has a similar edge to uses and initialof as the syntax for the goto graph edge would be: $(v_A, v_B, totalises, \succ)$. The allows relation has a weak textual order where the goto edge syntax would be $(v_A, v_B, allows, \preceq)$. The requires relation uses the textual symbol '$\leftrightarrow$' as the requires describes a relation that is *subpart of*, for example if we have v_A, requires, v_B then this describes a relation that v_A is *subpart of* v_B therefore the goto edge syntax would be $(v_A, v_B, requires, \leftrightarrow)$.

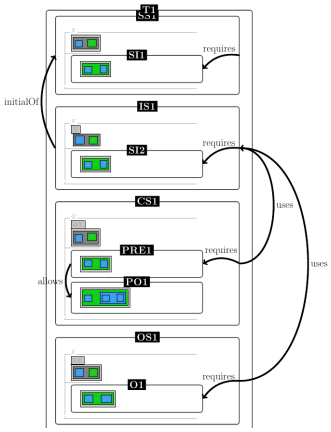


Figure 8.8: All annotations from table 8.2 combined into one specification.

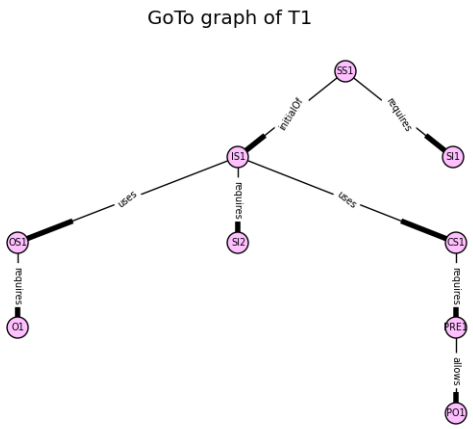


Figure 8.9: Goto graph of the example in 8.8

Therefore our nodes would be:

$\mathbb{V}_{figure\ 8.9} = \{$

$(T1, theory), (SS1, stateSchema), (SI1, stateInvariants),$

$(IS1, initSchema), (SI2, stateInvariants), (CS1, changeSchema),$

$(PRE1, precondition), (PO1, postcondition), (OS1, outputSchema), (O1, output)\}$

and our edges would be:

$\mathbb{O}_{figure\ 8.9} = \{$

$((SS1, stateSchema), (SI1, stateInvariants), requires, \leftrightarrow)$

$((IS1, initSchema), (SI2, stateInvariants), requires, \leftrightarrow)$

$((IS1, initSchema), (SS1, stateSchema), initialOf, \succ)$

$((CS1, changeSchema), (PRE1, precondition), requires, \leftrightarrow)$

$((PRE1, precondition), allows, (PO1, postcondition), allows, \preceq)$

$((CS1, changeSchema), (IS1, initSchema), uses, \succ)$

$((OS1, outputSchema), (O1, output), requires, \leftrightarrow)$

$((OS1, outputSchema), (SS1, stateSchema), uses, \succ)$

$\}$

Therefore the goto graph would be the pair $\mathbb{V}_{figure\ 8.9}, \mathbb{O}_{figure\ 8.9}$ which we could use to input into a theorem prover.

8.4.1 Conclusion

In this section we took a formal view of the GoTo graph. We have described the edges as a quadruple and defined the graph itself to be a pair. The textual order of each relation has been described and an example has been shown of how to define a goto graph formally.

8.5 Chapter Conclusion

Since the ZCGa has already been touched upon formally via [40] it was important to somewhat formally define the ZDRa and graphs produced. This chapter took

a formal view on the ZDRa and described the aspect using certain definitions and rules to be followed. Formal properties about the aspect have been highlighted and proved using the definitions and rules defined throughout this thesis.

The automatically generated dependency graph has also been looked at formally using definitions for the nodes and edges. Examples have been given of formally defined specifications using the definitions highlighted. We took a look at how different symbols and facts are introduced using instances annotated in ZMathLang.

The generation of the Goto graph from the dependency graph was illustrated in detail. The textual order of the relations was then described and explained why it was important. Examples were given of dependency graphs and goto graphs to highlight how they varied and the algorithm of the goto and dependency generation was given.

The consecutive chapter highlghted the formalities of the goto graph and gave formal definitions similar to the formal take on the dependency graph. The relations were given certain rules to follow to describe the textual order of them and then an example was given.

In the next chapter we take a look at the interface in which a one can use the ZMathLang and ultimatly translate a formal/semi formal specification into a theorem prover.

Chapter 9

Interface

The interface was designed so that the steps to convert a Z specification to a fully proven specification would be easier to complete by the user. It made the ZMathLang process more user friendly then by just typing commands in a terminal. Full details for the user interface and all its functions can be found in Mihaela's user guide [59]. This chapter only explains the process needed to translate a specification into a full proof, other steps such as writing the specification in the first place and outputting pdf using the interface can be found in the manual.

9.1 Inserting a specification

To use the ZMathLang framework through the interface the user can download the files from [14] then using a terminal run the interface program by typing `python Interface.py`, figure 9.1 shows an example of this.

```
jeff@jeff-laptop:~/lavinias_workspace/example$ python Interface.py
```

Figure 9.1: Example of how to start the interface for the ZMathLang framework.

An example of the interface can be seen in figure 9.2. Depending on the operating system the interface may look slightly different however the main panels and buttons will be the same. The main menu bar is at the top left had side and the main panel is on the left. There is a messages panel on the right top side which displays any messages when checking for correctness and converting to skeletons. To check a

specification the user can click *file* then *open* from the main menu as shown in figure 9.2.

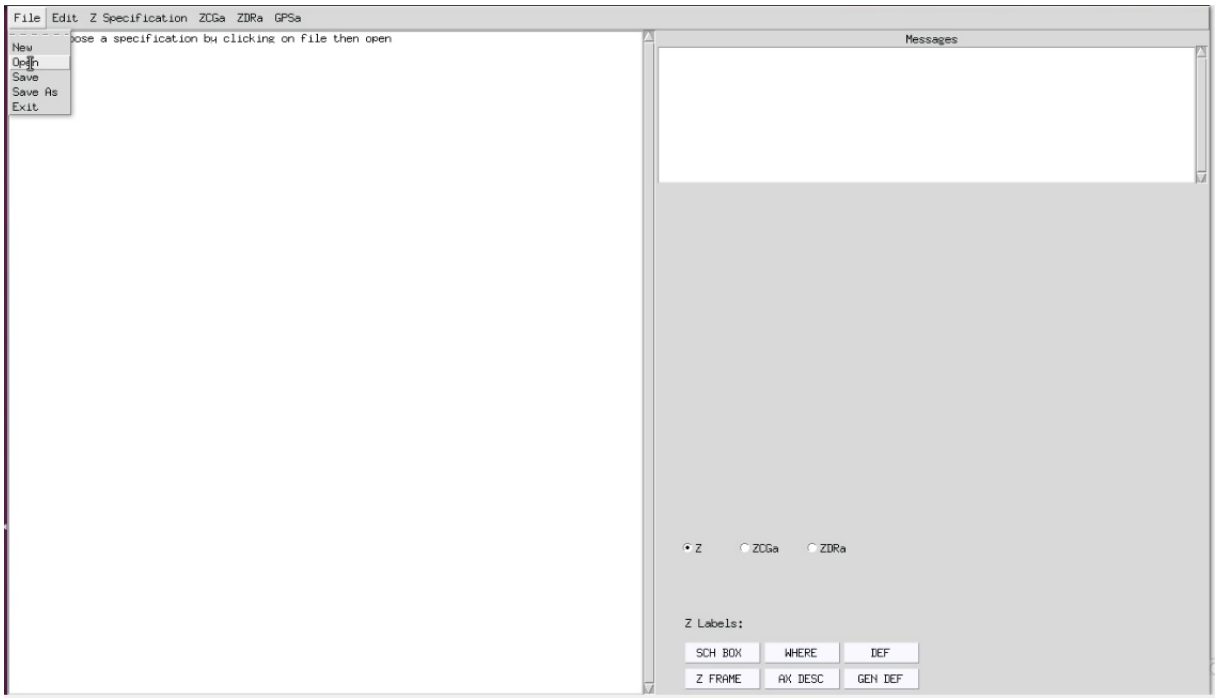


Figure 9.2: Example of the interface and opening a specification.

A pop up box appears asking the user to locate the specification they would like to translate. An example of the pop up box is shown in figure 9.3.

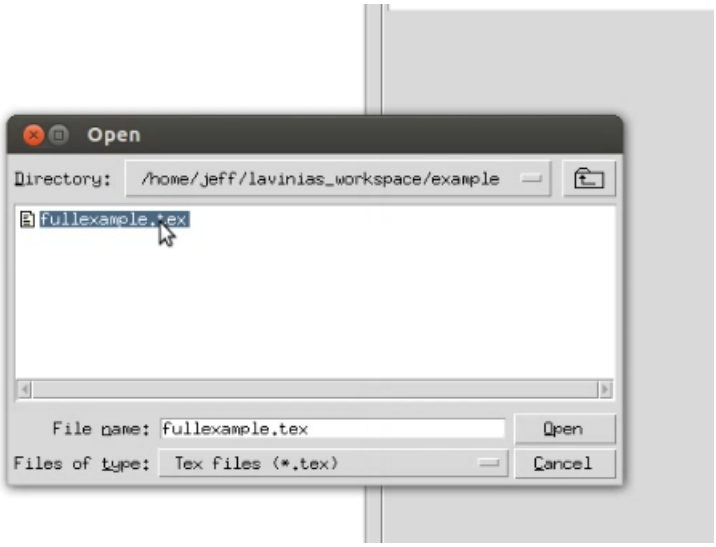


Figure 9.3: Asking the user to insert the specification.

Once a specification has been chosen then the specification should appear in the

panel on the left hand side. Figure 9.4 shows an example of this. Note that no messages appear yet in the messages box.

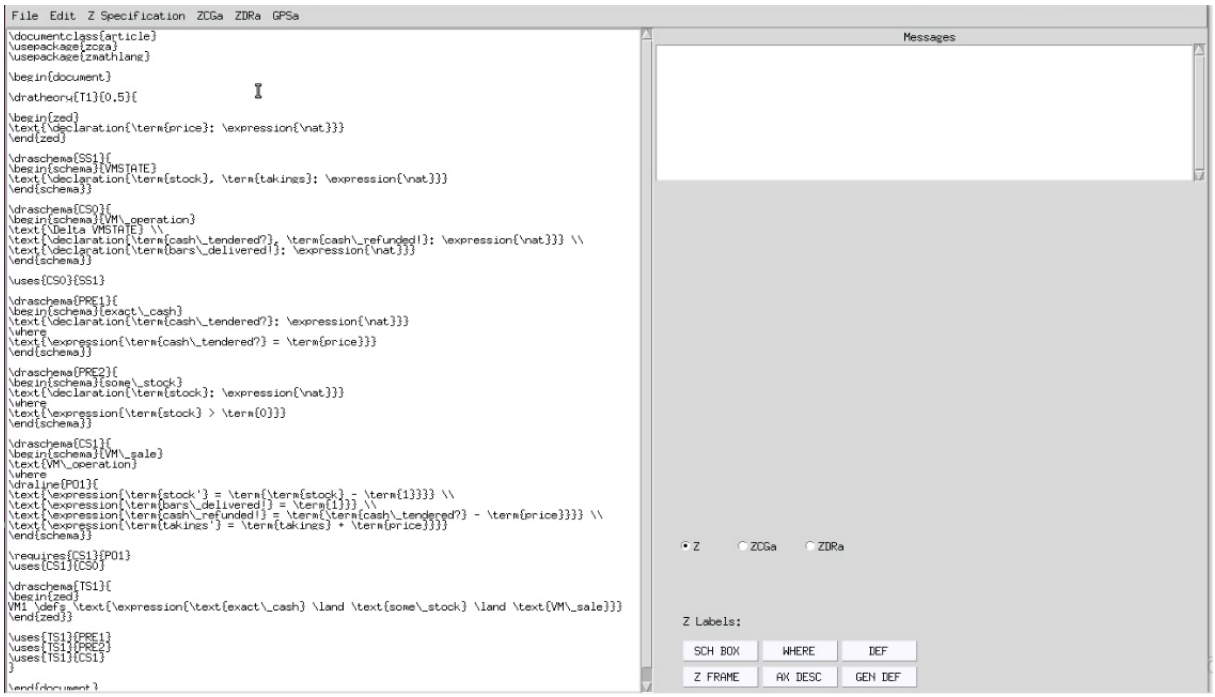


Figure 9.4: Example of a specification inserted in the main panel.

9.2 Checking ZCGa

To then check for ZCGa correctness the specification loaded into the interface must be ZCGa annotated.

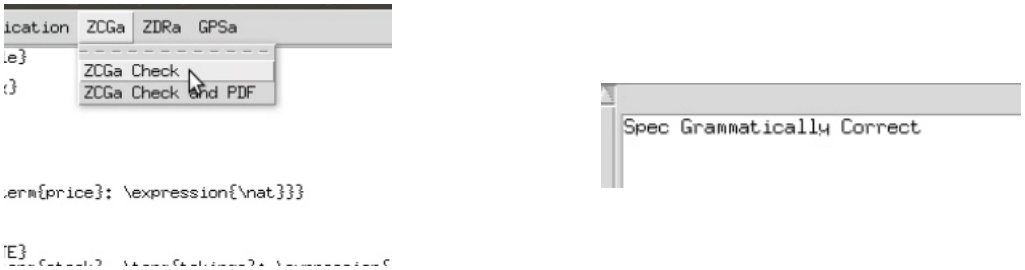


Figure 9.5: An example of how to check the specification for ZCGa correctness (left) and the message which appears when the specification is ZCGa correct.

To check for ZCGa correctness the user can click on the *zcga* button from the top menu and then click on *Zcga Check*. If the specification is grammatically correct

then a message appears in the message box in the top right of the interface (n see figure 9.5).

9.3 Checking ZDRa

To check for ZDRa correctness the specification loaded into the interface must be labelled with ZDRa annotations (this can be with or without ZCGa annotations).

Figure 9.6: An example of how to check the specification for ZDRa correctness (left) and the message which appears when the specification is ZDRa correct.

To check for ZDRa correctness the user can click on the *ZDRa* button in the top menu of the interface and then on the *Zdra Check* button. If the specification is ZDRa correct and the specification has been correctly totalised then a message confirming this appears in the message box. Figure 9.6 shows both of these actions.

9.4 Skeletons

The user may also want to create a general proof skeleton, Isabelle skeleton and fill in the Isabelle skeleton using the Interface. This section explains how this may be done.

9.4.1 General Proof Skeleton

To create a general proof skeleton from the users ZDRa annotated and correct specification. The user will need to input their specification into the interface, check the specification for ZDRa correctness then click on the *GPSa* button in the top menu and choose *Proof Skeleton* fromt the drop down menu. An example of this is shown in figure 10.11.

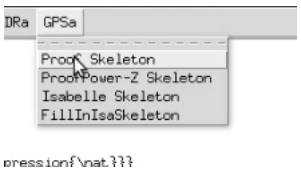


Figure 9.7: How to create a general proof skeleton using the user interface.

A new file should then appear in the same folder as your `Interface.py` program (see figure 9.8).

Figure 9.8: New general proof skeleton.

The user may then open this file using an external text editor or they may view it in the interface itself. When creating a general proof skeleton a message appears in the Messages box saying `Skeleton Created`, a button also appears underneath the message box saying `Show Skeleton` (see figure 9.10). By clicking on this new button the general proof skeleton can be opened within the Interface.

9.4.2 Isabelle Skeleton

After creating a general proof skeleton the user may want to take the translation one step further and create an Isabelle proof skeleton. Again, to do this the specification must be labelled with ZDRa and be ZDRa correct. The user may then click on the *GPSa* button in the top menu and then the *Isabelle Skeleton* button in the drop down menu as shown in figure 9.9.

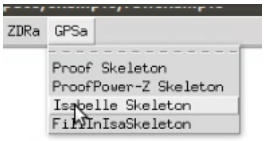


Figure 9.9: The user choosing to create an Isabelle skeleton in the user interface.

If all is correct a message saying `Isabelle Skeleton Created` and a new button should appear under the message box with the text `Show Isabelle Skeleton` as

seen in figure 9.10. A new file will be produced and automatically saved in the same directory as the interface with a *.thy* extension.

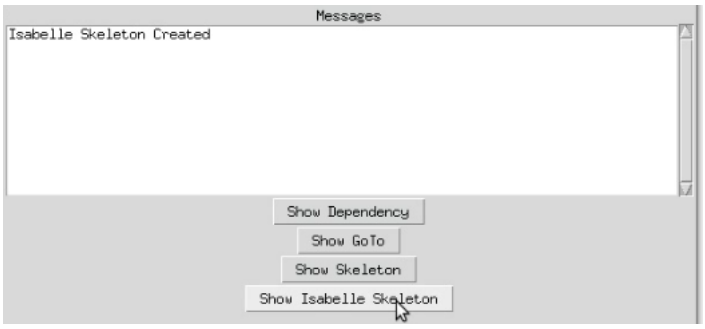


Figure 9.10: New buttons which appear if the user chooses to create a general proof skeleton or an Isabelle skeleton.

The user may then open the Isabelle skeleton using an external text editor such as Jedit or Isabelle itself or they may choose to open the Isabelle Skeleton within the interface by clickon on the `Show Isabelle Skeleton Button`. An example of a pop-up window showing the Isabelle skeleton in the user interface can be seen in figure 9.11.

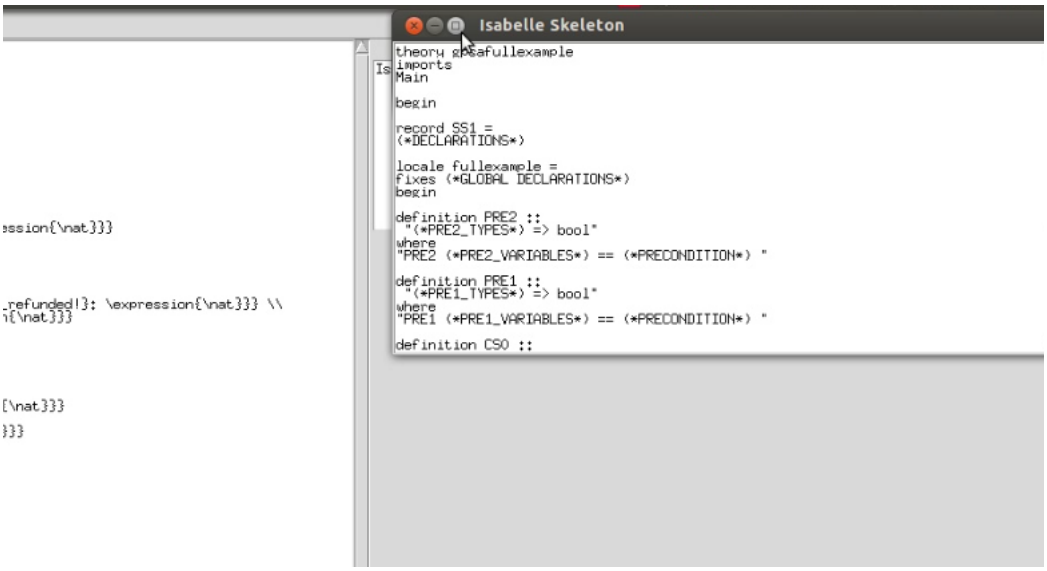


Figure 9.11: Popup box in the user interface showing the isabelle skeleton.

The user may even wish to go as far as filling in the Isabelle skeleton. To do this the Isabelle skeleton will need to be created first and the specification loaded into the interface must also be annotated with ZCGa. The user can then click on `GPSa` in the top level menu and then `FillInIsaSkeleton` (see figure 9.12). If the skeleton

is not in the same directory as the interface or has been renamed then the interface will ask for the user to locate the Isabelle skeleton in a similar way to opening a specification (figure 9.3).

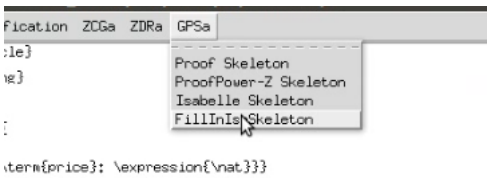


Figure 9.12: The user choosing to fill in the isabelle skeleton in the user interface .

The user may then wish to open the filled-in isabelle skeleton externally or in the interface the same was as they opened the skeleton in the previous step.

9.5 Output messages which could occur

Sometimes there is an error in any of the actions previously described in this chapter. The message box not only tells the user what has been successful but it also gives the user information if there has been some error in the action they were trying. Table 9.1 shows all the possible messages which can appear in the message box along with an explanation about the message.

Text in message box	**Explanation**
`Specification Correctly Totalised`	All preconditions in the specification have a totalising condition
`Warning! Specification not correcty totalised`	Not all preconditions have an alternative output (skeleton can still be created) *(see chapter **??**)*
`Specification is Rhetorically Correct`	Specification is ZDRa correct *(see chapter **??**)*
`Skeleton Created`	General proof skeleton has been successfully created
`Isabelle Skeleton Created`	Isabelle skeleton has been successfully created
`Isabelle Skeleton successfully filled in`	Isabelle proof skeleton has been filled in using ZCGa text
`Please convert specification into Isabelle Skeleton first`	Covert the specification into an Isabelle skeleton first before filling it in *(see chapter 6)*
`Please select your isabelle skeleton:`	Please locate the Isabelle skeleton which you wish to be filled in *(see chapter 6)*
`Please convert specification into GPSa first`	Can not find the general proof skeleton *(see chapter 6)*
`Loops in reasoning Can not create Skeleton`	Specification is not ZDRa correct and the skeleton can not be created *(see chapter 5)*

`Spec Grammatically Correct`	The specification has passed ZCGa checker
`Spec Grammatically Incorrect` `Number of errors: 2`	The specification has failed ZCGa correctness and has 2 errors *(see chapter* ??*)*

Table 9.1: Messages which could appear in the user interface and their meanings.

9.6 Conclusion

This chapter has described how a user of the ZMathLang framework can use the implemented user interface to assist them with checking for grammatical and rhetorical correctness. The user interface also gives a clear and easy way to translate the specification into a general proof skeleton, Isabelle skeleton and filling in the Isabelle skeleton without using the user unfriendly terminal. The interface also allows the user to view the documents automatically produced from the annotated specification.

Chapter 10

From raw specification to fully proven spec: A full example

In this chapter we take a system specification through all the steps of ZMathLang to demonstrate how we can get from a raw specification to a full proof. We have added commentary throughout so the reader can understand how we can get from one step to another. We have added figures and screenshots of each step of the ZMathLang framework.

10.1 Step 0

Raw Specification

We take a raw specification ModuleReg [22], which displays students taking modules in a school environment (shown in figure 10.1) The output for the specification can be seen in figure 10.2. The full input for this can be found in appendx **??**.

```
\documentclass{article}
\usepackage{zmathlang}
\begin{document}

\begin{zed}
[PERSON, MODULE]
\end{zed}

\begin{schema}{ModuleReg}
students: \power PERSON \\
degModules: \power MODULE \\
taking: PERSON \rel MODULE
\where
\dom taking \subseteq students \\
\ran taking \subseteq degModules
\end{schema}

\begin{schema}{AddStudent}
\Delta ModuleReg \\
p?: PERSON \\
\where
p? \notin students \\
students' = students \cup \{p?\} \\
degModules' = degModules \\
taking' = taking
\end{schema}

\begin{schema}{RegForModule}
\Delta ModuleReg \\
p?: PERSON \\
m?: MODULE
\where
p? \in students \\
m? \in degModules \\
p? \mapsto m? \notin taking \\
taking' = taking \cup \{p? \mapsto m?\} \\
students' = students \\
degModules' = degModules
\end{schema}
\end{document}
```

Figure 10.1: Part of the raw schema.

ModuleReg

$$students : \mathbb{P}\, PERSON$$
$$degModules : \mathbb{P}\, MODULE$$
$$taking : PERSON \leftrightarrow MODULE$$
$$\text{dom}\, taking \subseteq students$$
$$\text{ran}\, taking \subseteq degModules$$

AddStudent

$$\Delta ModuleReg$$
$$p? : PERSON$$
$$p? \notin students$$
$$students' = students \cup \{p?\}$$
$$degModules' = degModules$$
$$taking' = taking$$

RegForModule

$$\Delta ModuleReg$$
$$p? : PERSON$$
$$m? : MODULE$$
$$p? \in students$$
$$m? \in degModules$$
$$p? \mapsto m? \notin taking$$
$$taking' = taking \cup \{p? \mapsto m?\}$$
$$students' = students$$
$$degModules' = degModules$$

Figure 10.2: Part of a raw specification output.

10.2 Step 1

ZCGa

The user then goes to label the raw specification with the ZCGa labels. The labelled specification can be seen in figure 10.3. The words highlighted in red are the ZCGa annotations done by the user and the black text is the existing specification. Figure 10.3 outputting result can be seen in figure 10.4.

```
\documentclass{article}
\usepackage{zmathlang}
\begin{document}

\begin{zed}
\set{PERSON}, \set{MODULE}
\end{zed}

\begin{schema}{ModuleReg}
\text{\declaration{\set{students}:\expression{\power PERSON}}}\\
\text{\declaration{\set{degModules}:\expression{\power MODULE}}}\\
\text{\declaration{\set{taking}:\expression{PERSON \rel MODULE}}}
\where
\text{\expression{\set{\dom \set{taking}} \subseteq \set{students}}}\\
\text{\expression{\set{\ran \set{taking}} \subseteq \set{degModules}}}
\end{schema}

\begin{schema}{AddStudent}
\text{\Delta ModuleReg}\\
\text{\declaration{\term{p?}:\expression{PERSON}}}\\
\where
\text{\expression{\term{p?} \notin \set{students}}}\\
\text{\expression{\set{students'} = \set{\set{students} \cup
\set{\{\term{p?}\}}}}}\\
\text{\expression{\set{degModules'} = \set{degModules}}}\\
\text{\expression{\set{taking'} = \set{taking}}}
\end{schema}

\begin{schema}{RegForModule}
\text{\Delta ModuleReg}\\
\text{\declaration{\term{p?}: \expression{PERSON}}}\\
\text{\declaration{\term{m?}: \expression{MODULE}}}
\where
\text{\expression{\term{p?} \in \set{students}}}\\
\text{\expression{\term{m?} \in \set{degModules}}}\\
\text{\expression{\term{\term{p?} \mapsto \term{m?}}
\notin \set{taking}}}\\
\text{\expression{\set{taking'} = \set{\set{taking} \cup
\set{\{\term{\term{p?} \mapsto \term{m?}}\}}}}}\\
\text{\expression{\set{students'} = \set{students}}}\\
\text{\expression{\set{degModules'} = \set{degModules}}}
\end{schema}
\end{document}
```

Figure 10.3: Part of the raw schema.

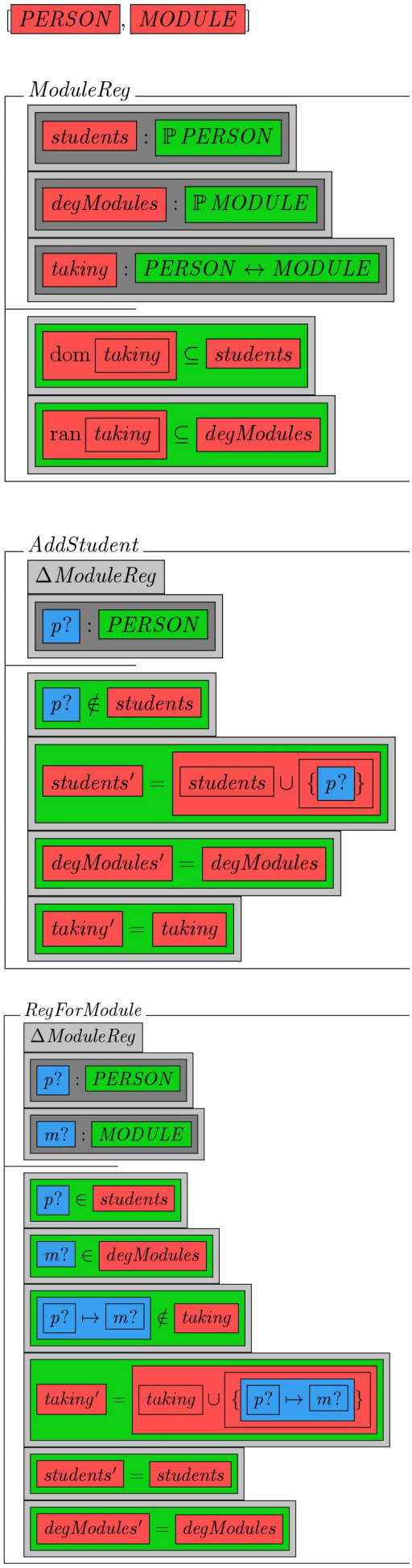


Figure 10.4: Part of a ZCGa labelled

After annotating we run it through the ZCGa correctness checker. Figure 10.5 shows the message which appears when the annotated specification has been checked.

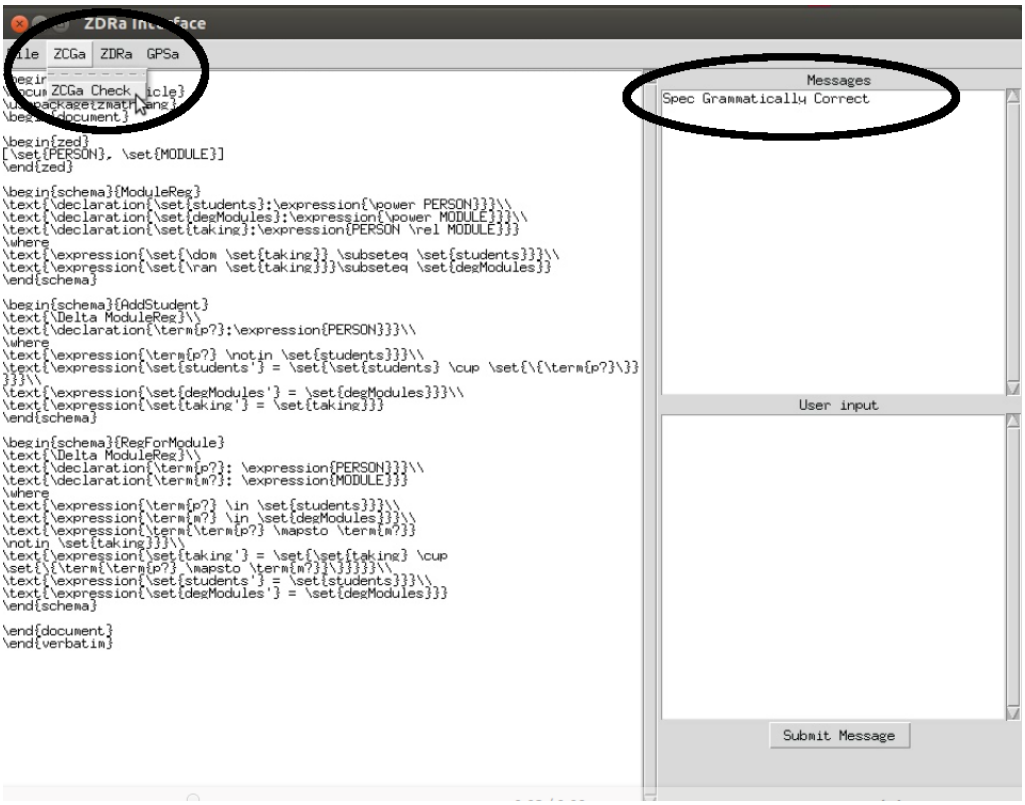


Figure 10.5: Message which appears after running the ZCGa checker on our example.

10.3 Step 2

ZDRa

Next the user can add ZDRa relations to chuk parts of the specification together and add relations to them. Figure 10.6 shows our example labelled in ZDRa annotations (in blue), the ZCGa annotations are in grey and existing specification in black. Figure 10.7 shows the compiled result.

```
\documentclass{article}
\usepackage{zmathlang}
\begin{document}

\dratheory{T1}{0.5}{
\begin{zed}
\set{PERSON}, \set{MODULE}
\end{zed}

\draschema{SS1}{
\begin{schema}{ModuleReg}
\text{\declaration{\set{students}:\expression{\power PERSON}}}\\
\text{\declaration{\set{degModules}:\expression{\power MODULE}}}\\
\text{\declaration{\set{taking}:\expression{PERSON \rel MODULE}}}
\where
\draline{SI1}{
\text{\expression{\set{\dom \set{taking}} \subseteq \set{students}}}\\
\text{\expression{\set{\ran \set{taking}} \subseteq \set{degModules}}}}
\end{schema}}

\requires{SS1}{SI1}

\draschema{CS1}{
\begin{schema}{AddStudent}
\text{\Delta ModuleReg}\\
\text{\declaration{\term{p?}:\expression{PERSON}}}\\
\where
\draline{PRE1}{
\text{\expression{\term{p?} \notin \set{students}}}}\\
\draline{PO1}{
\text{\expression{\set{students'} = \set{\set{students} \cup
\set{\{\term{p?}\}}}}}\\
\text{\expression{\set{degModules'} = \set{degModules}}}\\
\text{\expression{\set{taking'} = \set{taking}}}}
\end{schema}}

\requires{CS1}{PRE1}
\allows{PRE1}{PO1}
\uses{CS1}{SS1}

\draschema{CS2}{
\begin{schema}{RegForModule}
\text{\Delta ModuleReg}\\
\text{\declaration{\term{p?}: \expression{PERSON}}}\\
\text{\declaration{\term{m?}: \expression{MODULE}}}
\where
\draline{PRE2}{
\text{\expression{\term{p?} \in \set{students}}}\\
\text{\expression{\term{m?} \in \set{degModules}}}\\
\text{\expression{\term{\term{p?} \mapsto \term{m?}}
\notin \set{taking}}}}\\
\draline{PO2}{
\text{\expression{\set{taking'} = \set{\set{taking} \cup
\set{\{\term{\term{p?} \mapsto \term{m?}}\}}}}}\\
\text{\expression{\set{students'} = \set{students}}}\\
\text{\expression{\set{degModules'} = \set{degModules}}}}
\end{schema}}
}
\end{document}
```

Figure 10.6: An example of a specification labelled in ZCGa and ZDRa.).

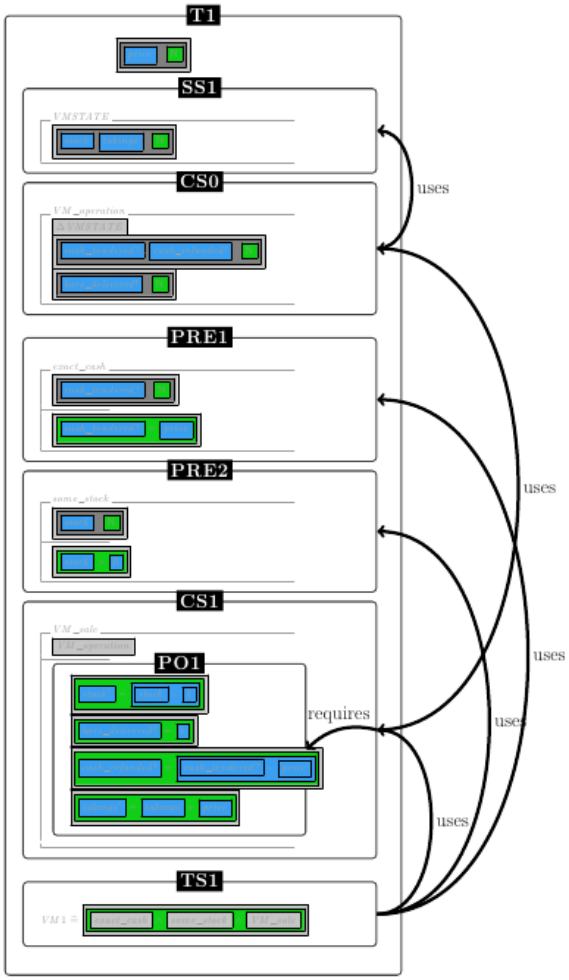


Figure 10.7: An example of a specification output labelled in ZCGa and ZDRa.).

After annotating our example in ZDRa labels we can then run our specification through the ZDRa checker. Figure 10.8 shows the message which appears after we check our example with ZDRa.

Figure 10.8: Message which appears after running the ZDRa checker on our example.

10.4 Step 2.5

Graphs

Since the example is ZDRa correct the two graphs shown in figures 10.9 and 10.10 are automatically produced and saved on the users computer.

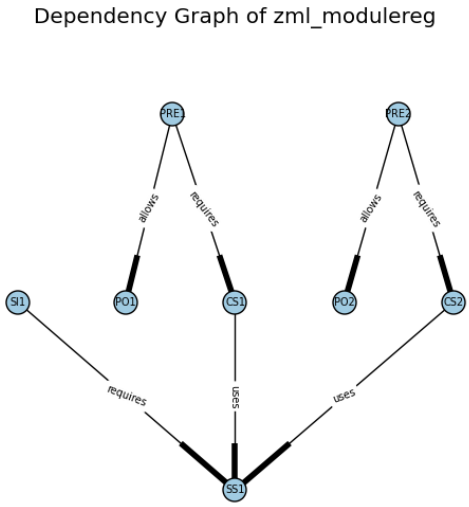


Figure 10.9: Dependency graph automatically generated from the ZDRa for our example.

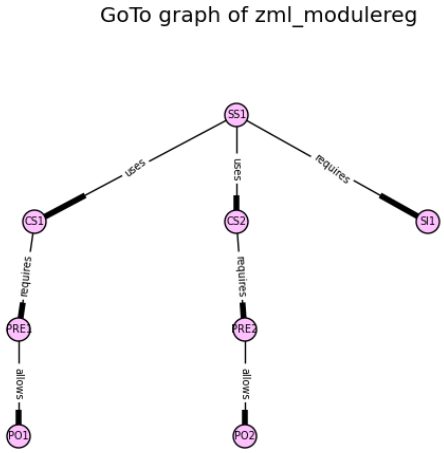


Figure 10.10: GoTo graph automatically generated from the ZDRa for our example.

10.5 Skeletons

The skeletons are automatically generated if the specification passes the ZCGa and ZDRa check.

10.5.1 Step 3

General Proof Skeleton

We can generate a general proof skeleton which prints out the ZDRa name and the instances they should be converted to when inputting into any theorem prover. If the specification is ZDRa] correct we can then generate the GPSa by clicking on the GPSa menu in the interface (figure 10.11).

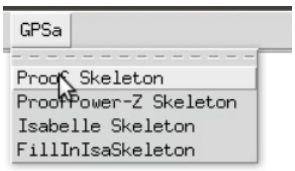


Figure 10.11: The GPSa button the interface which allows the user to generate the general proof skeleton.

Figure 10.12 shows the general proof skeleton which was generated for our example. Note all instances but the last 2 are actual instances labelled by the user. Since thre are 2 instances where there could be a change in state (CS1 and CS2) then there are 2 proof obligations added to the GPSa (L1_CS2 and L2_CS1).

```
stateSchema SS1
stateInvarients SI1
changeSchema CS2
precondition PRE2
changeSchema CS1
precondition PRE1
postcondition PO2
postcondition PO1
lemma L1_(CS2)
lemma L2_(CS1)
```

Figure 10.12: General proof skeleton.

10.5.2 Step 4

Isabelle Skeleton

From GPSa, the ZMathLang program can automatically generate an Isabelle Skeleton. The user can do this by clicking on the GPSa menu on the interface then

clicking '`Isabelle Skeleton`' shown in figure 10.13.

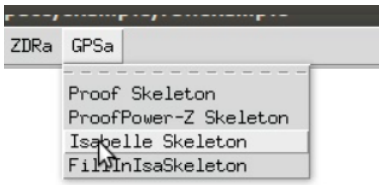


Figure 10.13: The Isabelle skeleton button the interface which allows the user to generate an Isabelle skeleton of their specification.

The Isabelle skeleton consists of the information generated in the general proof skeleton along with the environment to begin an Isabelle theory. It contains comments inbetween `(* .. *)` paranthesis to show the parts which need to be filled in either by using the ZCGa document or by the user. Figure 10.14 shows the automatically generated Isabelle skeleton for our `modulereg` example.

```
theory gpsa1n2
imports
Main
begin
(*DATATYPES*)
record SS1 =
(*DECLARATIONS*)
locale 1n2 =
fixes (*GLOBAL DECLARATIONS*)
assumes SI1
begin
definition CS2 ::
"(*CS2_TYPES*) => bool"
where
"CS2 (*CS2_VARIABLES*) ==
 (PRE2)
\<and> (PO2)"
definition CS1 ::
"(*CS1_TYPES*) => bool"
where
"CS1 (*CS1_VARIABLES*) ==
 (PRE1)
\<and> (PO1)"

lemma CS2_L1:
"(\<exists> (*CS2_VARIABLESANDTYPES*).
(PRE2)
\<and> (PO2)
\<and> (SI1)
\<and> (SI1'))"
sorry
lemma CS1_L2:
"(\<exists> (*CS1_VARIABLESANDTYPES*).
(PRE1)
\<and> (PO1)
\<longrightarrow> ((SI1)
\<and> (SI1')))"
sorry
end
end
```

Figure 10.14: Isabelle proof skeleton.

10.5.3 Step 5

Isabelle Skeleton Filled in

Using the ZCGa annotated document and the Isabelle skeleton described in the previous section. The user can then automatically fill in the missing information which is needed between the comment parenthesis `(* ...*)`. This is the final step which is automated by the program and the user can click on the GPSa button on the main menu bar in the inerface and then click on '`FillInIsa Skeleton`' in the sub menu (shown in figure 10.15).

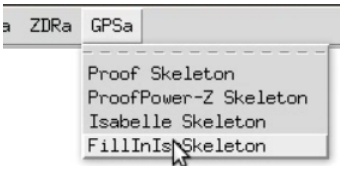


Figure 10.15: The Fill in Isabelle Skeleton button the interface which allows the user to fill in the skeleton they previously created.

Figure 10.16 shows our example with a filled in Isabelle skeleton. It is important to note that the program also changes some of the syntax from LaTeX to Isabelle so that it is fully parsable by Isabelle.

```
theory new5
imports
Main

begin
typedecl PERSON
typedecl MODULE

record ModuleReg =
STUDENTS :: " PERSON set"
DEGMODULES :: " MODULE set"
TAKING :: "(PERSON * MODULE) set"

locale gpsaModuleReg =
fixes students :: " PERSON set"
and degModules :: " MODULE set"
and taking :: "(PERSON * MODULE) set"
assumes "Domain taking \<subseteq> students"
 and "Range taking \<subseteq> degModules"
begin

definition RegForModule ::
"ModuleReg \<Rightarrow> ModuleReg \<Rightarrow>
PERSON \<Rightarrow> MODULE => MODULE set
\<Rightarrow> PERSON set \<Rightarrow>
(PERSON * MODULE) set \<Rightarrow> bool"
where
"RegForModule modulereg modulereg' p m
degModules' students' taking' ==
(p \<in> students)
\<and> (m \<in> degModules)
\<and> ((p, m) \<notin> taking)
\<and> (taking' = taking \<union> {(p, m)})
\<and> (students' = students)
\<and> (degModules' = degModules)"

definition AddStudent ::
"ModuleReg \<Rightarrow> ModuleReg => PERSON
\<Rightarrow>  MODULE set \<Rightarrow>
PERSON set \<Rightarrow> (PERSON * MODULE) set
\<Rightarrow> bool"
where
"AddStudent modulereg modulereg' p degModules'
students' taking' ==
 ((p \<notin> students)
\<and> (students' = students \<union> {(p)})
\<and> (degModules' = degModules)
\<and> (taking' = taking))"

lemma RegForModule_L1:
"(\<exists> degModules:: MODULE set.
\<exists> students :: PERSON set.
\<exists> taking :: (PERSON * MODULE) set.
\<exists> p :: PERSON.
\<exists> degModules':: MODULE set.
\<exists> students' :: PERSON set.
\<exists> taking' :: (PERSON * MODULE) set.
\<exists> m :: MODULE.
((p \<in> students)
\<and> (m \<in> degModules)
\<and> ((p, m) \<notin> taking)
\<and> (taking' = taking \<union> {(p, m)})
\<and> (students' = students)
\<and> (degModules' = degModules))
\<longrightarrow>
((Domain taking \<subseteq> students)
\<and> (Range taking \<subseteq> degModules)
\<and> (Domain taking' \<subseteq> students')
\<and> (Range taking' \<subseteq> degModules')))"
sorry

lemma AddStudent_L2:
"(\<exists> degModules:: MODULE set.
\<exists> students :: PERSON set.
\<exists> taking :: (PERSON * MODULE) set.
\<exists> p :: PERSON.
\<exists> degModules':: MODULE set.
\<exists> students' :: PERSON set.
\<exists> taking' :: (PERSON * MODULE) set.
 ( (students' = students \<union> {(p)})
\<and> (degModules' = degModules)
\<and> (taking' = taking))
\<longrightarrow
((Domain taking \<subseteq> students)
\<and> (Range taking \<subseteq> degModules)
\<and> (Domain taking' \<subseteq> students')
\<and> (Range taking' \<subseteq> degModules')))"
sorry
end
end
```

Figure 10.16: Filled In proof skeleton.

Figure 10.16 shows the original specification we started with in step 0 in Isabelle syntax. It is important to the reader to note that we have come this far without the user knowing any Isabelle at all. In our example we have 2 existing lemma's to prove to check the consitancy of the specification. That is the state before `RegForModule` (PRE2), *and* the state after `RegforModule` (PO2), *implies* `stateInvariants` (SI1), *and* the `stateInvariants'` (SI1') is true. So the precondition and postcondition imply that the stateInvariants and stateInvariants prime hold. The same goes for the `AddStudent` operation. When the skeleton is Filled in the ZMathLang is unable to prove the lemma's automatically and it is up to the user to do this (explanied in the next section). However at this stage ZMathLang puts the Isar command '`sorry`' as to ignore the lemma and act as if it was proven.

10.6 Step 6

Full Proof

The next part is to prove any existing lemmas from the filled in Isabelle Skelton or add new lemma's to prove safety properties about the specification. However, this final stage is difficult to automate with ZMathLang as everyone has different properties they wish to prove and all specification are different themselves. So the final step will need some theorem prover knowledge, but not as much as translating the specification and proving it in one step as the specification is already put into the theorem prover syntax. In this case the user may wish to use theorem prover tools which already exist such as Sledgehammer [9] to help them prove the properties. An example is shown in figure 10.17 the proof obligations being proven by hand for the `modulereg` specification.

```
lemma RegForModule_L1:
"(\<exists> degModules:: MODULE set.
\<exists> students :: PERSON set.
\<exists> taking :: (PERSON * MODULE) set.
\<exists> p :: PERSON.
\<exists> degModules':: MODULE set.
\<exists> students' :: PERSON set.
\<exists> taking' :: (PERSON * MODULE) set.
\<exists> m :: MODULE.
((p \<in> students)
\<and> (m \<in> degModules)
\<and> ((p, m) \<notin> taking)
\<and> (taking' = taking \<union> {(p, m)})
\<and> (students' = students)
\<and> (degModules' = degModules))
\<longrightarrow>
((Domain taking \<subseteq> students)
\<and> (Range taking \<subseteq> degModules)
\<and> (Domain taking' \<subseteq> students')
\<and> (Range taking' \<subseteq> degModules')))"

by (smt Domain_empty Domain_insert
Range.intros Range_empty Range_insert
Un_empty Un_insert_right empty_iff
empty_subsetI empty_subsetI insert_mono
insert_mono singletonI singletonI
singleton_insert_inj_eq' singleton_insert_inj_eq')

lemma AddStudent_L2:
"(\<exists> degModules:: MODULE set.
\<exists> students :: PERSON set.
\<exists> taking :: (PERSON * MODULE) set.
\<exists> p :: PERSON.
\<exists> degModules':: MODULE set.
\<exists> students' :: PERSON set.
\<exists> taking' :: (PERSON * MODULE) set.
((students' = students \<union> {(p)})
\<and> (degModules' = degModules)
\<and> (taking' = taking))
\<longrightarrow>
((Domain taking \<subseteq> students)
\<and> (Range taking \<subseteq> degModules)
\<and> (Domain taking' \<subseteq> students')
\<and> (Range taking' \<subseteq> degModules')))"

by blast
```

Figure 10.17: An example of a property and it's proof for the Vending Machine example.

Figure 10.17 shows an the lemmas automatically generated from the ZDRa annotated specification (black) and the user input needed to prove this lemma (red). Notice that the user has deleted the word "`sorry`" which would have automatically come after the lemma. More information on proving these lemmas can be found in the Isabelle Manual [77] and is beyond the scope of this thesis.

10.7 Conclusion

In this chapter we have taken a single specification and shown the entire path from the raw specification to it's translation in Isabelle. We have shown the LaTeX code and the compiled output for the raw specification, ZCGa annotated specification and ZDRa annotated specification. We have shown screenshots of the interface to demonstrate of how to check for each step of correctness. The dependency and goto

graphs where automatically demonstrated and displayed. Then the general proof skeleton, (which shows the order the instances must be in to input into a theorem prover) was displayed. We then generated an Isabelle proof skeleton for the vending machine and automatically filled it in using the ZMathLang program. In the final section we explained that it would be difficult to automate a proof due to the fact that the lemma's which need to be proved for a specification will vary due to the nature of the specification and the user who wishes to prove them.

In the next chapter we analyse the differences between translating a specification in one step and translating a specification using the ZMathLang method.

Chapter 11

Analysis

The concept for this thesis was to argue that breaking up the translation path from a formal specification to a full proof would be easier to conduct than to do a full proof all in one go. The vending machine example has been fully proved in PPZed and the birthday book example has been fully proved in Hol-Z. We will now look at these to examples and compare them to the proofs done in a stepwise method using MathLang.

It is important to note, the way the specifications are translated into Isabelle/Hol syntax is just one way. There are various other ways one may choose to translate specifications into Isabelle. Some of there other variations are described in [44], [45] and [13].

11.1 Vending Machine Example

The vending machine example shown in appendix **??** is a simple specification using only natural numbers as variables and there are no other types in the specification.

Method	**expertise required**	**input**	**lines of proof for first lemma (fl) entire proof (ep)**
One step into PPZed	much	Either ascii or windows extended characters	fl = 19 ep = 140
Multiple steps using ZMathLang	little	LaTeX partially automated into Isabelle	fl = 3 ep = 124 (63 automated)

Table 11.1: The vending machine proof using PPZed verses the ZMathLang proof.

Table 11.1 shows an outlined comparison between the vending machine proof done in PPZed [6] and the vending machine proof done using the ZMathLang method (see appendix A.1). To calculate the lines of proof, all comments and empty lines have been removed from the proof and only the content is left. Although the syntax of the proof can differ depending on the author, for example some of the tactics can be put on a single line or can be put on two separate ones, the lines of proof give a rough estimate in the size of proof.

The entire proof using the ZMathLang method is 124 lines, 63 of those lines are automatically generated using the annotated LaTeX document (79 lines). This means that 50.8% of the proof is already automatically generated without the user having any knowledge of the theorem prover they are using. The actual amount of lines in both the proofs are somewhat similar (140 lines compared with 124).

Type of expertise needed	**one step into PPZed**	**multi step using ZMathLang**
Knowledge of Z	yes	yes
Knowledge of theorem prover	much	little
Knowledge of LaTeX (including Z symbols)	some (optional)	yes
Knowledge of how to input specification into theorem prover	yes	no

Table 11.2: Expertise needed for one step proof in PPZed and multi step proof using ZMathLang.

The expertise needed to do either proof is shown in table 11.2. Here we explain the different types of expertise needed in order to get the vending machine specification into a full proof using one step or using many steps.

11.1.1 Knowledge of Z

For both methods the user will need to have some form of Z specification knowledge. Using the ZMathLang method, the user also annotates the plain specification which would then in turn allow others (such as staff in the project team, software developers, etc) also understand the Z specification. Both methods need the same amount of expertise in Z and the ZMathLang method even shares some of the expertise with others looking at the documents produced.

11.1.2 Knowledge of theorem prover

In table 11.2, it states that a "little" amount of knowledge of theorem prover is needed for the full proof using ZMathLang. This is because the final step is to prove any lemmas that are left unproven (these lemmas have been created from the original Z specification) and write new properties as lemmas and prove them

if needed. However the original specification is automatically translated into your chosen theorem prover syntax and thus if the user needs to add more parts to the specification they already have an idea of the syntax to use. By translating the specification and proving it in one big step the user will need to learn how to input the specification first, as well the syntax of a specification in the chosen theorem prover language and write up a full proof.

11.1.3 Knowledge of LaTeX

The translation path using the ZMathLang methods assumes the user already knows how to write a Z specification using LaTeX. The user then annotates these specifications using the annotations in the ZMathLang style package. The LaTeX expertise required for the translation is enough so the whole Z specification is covered. The input of the schema boxes, Z characters etc are all imputed using the zed style package, which the user can learn using Mike Spivey's reference card [72].

The schema boxes and symbols are written in PPZed's own syntax. PPZed also has a user interface, (PPXPP), in which it uses an extended character set instead of ascii to input the specifications and their proofs. In this, the user may open a palette in which they can search for the symbol they wish to use and click on it. The same works with schema boxes, axiomatic definitions, generic definitions etc.

The translation method using PPZed requires some LaTeX knowledge which is optional. This is only if the user wishes to extract the formal material for typesetting their proofs. The shell script **doctex** allows the user to prepare a LaTeX file using the PPZed extended character set. However to typeset the proof the instructions say that some familiarity with LaTeX is required.

11.1.4 Knowledge of input of specification

Discounting the tactics and lemmas needed to prove the specification. A large part of full proof for the specification is to input the specification itself into the chosen theorem prover. By translating the specification in one big step using PPZed the user must already have vast knowledge of PPZed to do this. That is, to translate

the specification itself in one big step into **any** theorem prover requires a lot of knowledge about the chosen theorem prover. By using the ZMathLang method to translate the specification itself requires no knowledge about Isabelle by the user, as all the Isabelle syntax is automatically translated from the annotated specification written in LaTeX.

11.2 Birthday Book Example

The birthday book example (shown in appendix app:bb), was created by Spivey [71]. This example is a specification which describes a system of a birthday book where the main functions include adding a person and their birthday, removing a person and their birthday etc. This system uses sets and it's own types, `NAME` and `DATE`.

Method	**expertise required**	**input**	**lines of proof for first lemma (fl) entire proof (ep)**
Hol-Z	some	automated into ZeTa, manually into Hol-Z	fl = 5 ep = 361
Multiple steps using ZMathLang	little	LaTeX partially automated into Isabelle	fl = 8 ep = 120

Table 11.3: The birthday book proof using Hol-Z verses the ZMathLang proof.

Table 11.3 shows the comparison between the birthday book proof done using Hol-Z [13] and the birthday book proof done using the ZMathLang method (see appendix 11.3). Again to calculate the lines of proof, all comments and empty lines have been removed from the proof. Since the birthday book proof in Hol-Z comes in many different files, all the lines from these files have been added. The translation via ZMathLang translates to Isabelle using just the `Main` isabelle package.

The first lemma (fl) in the table has been calculated from the "pre addBirthday lemma" which is called `lemma AddBirthdayIsHonest` in the ZMathLang method and `zlemma lemma2` in the *Rel_Refinement.thy* file using the Hol-Z method. The full proof using the Hol-Z method is 361 lines, however this is split up into 5 files. The *BBSpec.holz* which is automatically generated using the ZeTa-to-Hol-Z converter. This converted consists an adapter that is plugged into ZeTa and converts the LaTeX specification into an SML-file. The *BB.thy* file which is used to import *Fun_Refinement.thy* and *Rel_Refinement.thy* and *BBSpec.thy* which is used to import the specification from the SML-file. In order to prove the specifications in Hol-Z, there are 17 other theory files which have been created in order to use tactics an lemmas, these include *ZSeq.thy*, *Z.thy*, *ZPure.thy*.

The raw LaTeX file used for the Hol-Z method is 97 lines, this is automatically generated into an SML file which can be imported into Hol-Z which is 17 lines long. The raw LaTeX file which is used for the ZMathLang method is 96 lines which automatically generates a single theory file containing the environment and the specification, this file is 70 lines.

Type of expertise needed	**large steps into Hol-Z**	**small steps using ZMathLang**
Knowledge of Z	yes	yes
Knowledge of theorem prover	some	little
Knowledge of LaTeX	yes	yes
Knowledge of how to input specification into theorem prover	some (sml into Hol-Z)	no

Table 11.4: Expertise needed for one step proof in PPZed and multi step proof using ZMathLang.

Table 11.4 shows the type and amount of expertise needed in order to get from a specification into a fully proof.

11.2.1 Knowledge of Z

For both methods the user will need to have some knowledge of Z specifications. This is because in both methods the initial step is to write the specification in LaTeX for the system. However by using the ZMathLang method when annotating the Z specification in ZCGa, toe compiled documents outputs the weak types in different grammars. This then allows others to identify certain parts of the Z syntax. Therefore the knowledge of Z is exactly the same in both these methods.

11.2.2 Knowledge of theorem prover

Table 11.4 shows that by using the ZMathLang method 'little' knowledge of theorem prover is needed. This is because the final step to prove any unproven lemmas and write and new safety properties and lemmas and prove them. For this the user may need some theorem prover knowledge to compete this final step. However the entire specification itself is already written in the chosen theorem prover (in our case Isabelle) and the user does not need to import any further definitions which are part of the original specification. However, by using Hol-Z method the user will need 'some' theorem prover knowledge. Although it is possible to ease the translation of the specification into Hol-Z using ZeTa (see section 11.2.4), the user will need to have the Hol-Z plugin knowledge as well as the original Isabelle/Hol Knowledge to do the proofs for the specification.

11.2.3 Knowledge of LaTeX

In both methods the user will need to have the same amount of knowledge of LaTeX. This is because in both cases, the user will need to input their specification using LaTeX. The only difference in this aspect is that the user will need to annotate their specification using ZMathLang annotations (ZCGa and ZDRa) in the ZMathLang method or the user will need to annotate their specification using Hol-Z annotations (proof obligations, zsections etc). In both cases the user will need to know how to import a package into LaTeX and then read the instructions in either case on the annotations which need to be used.

11.2.4 Knowledge of input of specification

When translating Z specification into the Hol-Z theorem prover there are two ways a user can do this. The first method for convenience, involves the user writing their specification in LaTeX, using the Hol-Z package to annotate their specification. Then the ZeTa-to-Hol-Z plug-in type checks the specification and generates .holz files which can be imported by the user into the Hol-Z theorem prover. The method is have the user write the specification directly into Hol-Z circumventing ZeTa. In both these methods the user would need at least some form of Hol-Z prover knowledge. The latter would need more than the former. By using the ZeTa-to-Hol-Z plug-in, the user can write their specification in LaTeX format with the Hol-Z annotations (very similar to ZMathLang method), however the user will need to know how to import the .holz files into the Hol-Z theorem prover, unpack the schemas and values, and how to write and prove the properties.

To input the specification into the chosen theorem prover using ZMathLang the user will need no theorm prover knowledge at all. This is because the annotated specification will be automatically translated into Isabelle/Hol when using the ZMathLang method. The program will automatically generate an '.thy' file which is a skeleton of the specification and then automatically fill in the specification using the information from the ZCGa annotations.

11.3 Conclusion

This section compares 2 specifications written in Z which have been proven in a theorem prover previously with the proofs done using ZMathLang. The ZMathLang framework allows the user analyse their formal specification and assists them translating the specification itself into a theorem prover. However the last step of the framework to prove properties about the specification is still a difficult step in both translation paths (via ZMathLang or via another route). However the ZMathLang framework is there to give a helping hand to users who are complete beginners in proving formal specifications. Proving the actual properties and the proof obliga-

tions of the specification are a whole research area on their own and beyond the scope of this thesis but touched upon in chapter 2.

Even when teaching the syntax of Z in an academic setting, the ZMathLang aspects and it's tools can be used as helpful tools to help students understand the syntax of their formal specifications. Such as allowing students to annotate the grammatical categories ie what is a declaration, what is an expression. The students can also use the ZDRa instance to highlight which schema is a totaliseSchema, and which schema is a changeSchema etc.

In the next chapter we conclude our findings of this thesis and highlight what areas are of interest for future work.

Chapter 12

Evaluation and Discussion

In this chapter we go through a few case studies and discuss the difference between the specification translations if any. Table 12.1 shows the specifications we have translated into Isabelle using ZMathLang. We have classified these examples to show the different types of specifications which can be translated using the ZMathLang toolkit. In this chapter we take one example from each class and describe in more detail how the translation was done.

Examples using only terms	**Examples using sets and terms**
Vending Machine	Birthday Book
SteamBoiler	ClubState
Incomplete translations	Clubstate2
Autopilot	GenDB
A specification which fails ZCGa	ModuleReg
A specification which fails ZDRa	ProjectAlloc
	Timetable
	Videoshop
	TelephoneDirectory
	ZCGa

Table 12.1: A table showing the specifications we have translated into Isabelle using ZMathLang

We have categoriesed the specification into three groups; specifications which

only use terms, specifications which use both terms and set and specifications which the translation is incomplete for a variety of reasons. All the specifications we have translated are 'state based specifications', which means they operate within a state and to change the state their may become precondition and postconditions within the state. Some specifications are described differently such as functional specifications, however those type of specifications are out of the scope of this thesis.

12.1 Complexity of specifications

This section we analyse the complexity of the specifications we have translate using ZMathLang. First we check the complexity of the raw LaTeX specification file, without any annoatation. Then we discuss the complexity of the ZCGa annotated specifications and ZDRa annotated specifications and how this affects the translation into Isabelle.

12.1.1 Raw Latex Count

Table 12 how long each specification is by amount of lines of code and environments uses. We have listed the specifications in decreasing complexity of how many lines of LaTeX the raw specification has.

Specification	**Environment**				**Lines of LaTeX**
	Zed	Schema	Axdef	**Total**	
Steamboiler	10	34	3	47	507
ProjectAlloc	4	17	0	21	213
VideoShop	3	15	0	18	166
TelephoneDirectory	6	11	0	17	133
ClubState	4	11	1	16	129
ZCGa	2	9	0	11	128
GenDB	2	7	0	9	114
Timetable	1	6	1	8	92
BirthdayBook	3	7	0	10	83
AutoPilot	2	3	0	5	83
ClubState2	1	6	1	8	80
Vending Machine	4	7	0	11	68
ModuleReg	1	3	0	4	43

Table 12.2: How many zed, schema and axdef environments and lines of LaTeX code makes up each specification

We list information about how many different envronments and lines of LaTeX make up each specification in table 12.2. The environment numbers count how many different types of environments exist within the specification. That is how many '`\begin{schema}...\end{schema}`' or '`\begin{zed}`... etc. We add up the total amount of environments in the specification. From the table we can see that for most of the specifications the more lines of LaTeX there is then the total amount of environments increse. However, there are three exceptions to this trend. The '*BirthdayBook*' specification, '*ClubState2*' specification and '*Vendine Machine*' specification. Specifications for systems are can always be written in a variety of ways and still have the same meaning. Even formal specifications can be written different ways. For example one may have the following declarations:

$t : \mathbb{N}$

$l : \mathbb{N}$

However, this declaration can also be written as the following:

$t, l :; \mathbb{N}$

Thus removing a line. Formal specifications can also include comments written in natural language which are not part of the formal script. These extra comments about the specification may have also added to the line count in table 12.2.

12.1.2 ZCGa Count

In this section, we evaluate the ZCGa annotations on the specifications. We describe how many of each ZCGa annotations occurs for each specification we have translated.

Specification	**ZCGa WeakTypes**					
	□	■	■	■	■	■
Steamboiler	297	26	282	595	4	0
ProjectAlloc	98	43	113	154	165	0
VideoShop	87	31	75	119	95	0
TelephoneDirectory	78	26	53	72	50	0
ClubState	75	17	51	55	51	0
ZCGa	73	27	67	35	133	0
GenDB	45	24	71	117	121	1
Timetable	35	15	53	48	114	0
BirthdayBook	26	11	24	28	19	0
AutoPilot	16	9	19	31	2	0
ClubState2	34	7	37	22	72	0
Vending Machine	16	7	21	37	0	0
ModuleReg	20	6	18	13	31	0

Table 12.3: How many of each grammatical category exists in each specification.

The amount of times a ZCGa weak type occurs in each specification is shown in table 12.3. We remind the reader the colours corresponding to each grammatical type are: *schematext*, *declaration*, *expression*, *term*, *set* and *definition*. In this instance we don't use specification as we assume each document contains a single specification.

In our sample set we only have one specification (GenDB) with a '`definition`' annotation. This `definition` is locally defined within the specification. The '*Vending Machine*' specification only uses `terms` and therefore there are no ZCGa `term` annotations. However the '*StemBoiler*' specification also only uses term yet there are 4 `set` ZCGa annotations. This is because some of the `terms` used in the specification have to be intrduced by a `set`. For example in the *SteamBoiler* specification we have the following annotation:

```
\begin{zed}
\set{State} ::= \term{init} | \term{norm} |
\term{broken} | \term{stop}
\end{zed}
```

Although the set `State` is annotated as a set, it is not used in any of the schema's in the rest of the specification. It is only defined to present the terms `init`, `norm`, `broken` and `stop` which are used in the specification.

We expect there to be more `schemaText`'s then `declarations` and `expressions` combined as `schemaText` contains all `declarations`, `expressions` and SchemaNames however, from the table we can see that this is not always the case. For example in the *ProjectAlloc* example, there are 98 `schemaText`, 43 `declarations` and 113 `expressions`. The reason for this could be because a single `expression` can in itself contain many `expressions`. For example the following `schemaText` has been taken from the *ProjectAlloc* specification:

```
\text{\expression{\forall
\declaration{\term{lec}: \expression{\dom maxPlaces}}\\
@ \expression{\term{\# (\set{\set{allocation}
\rres \set{\{\term{lec}\}}})} \leq \term{\set{maxPlaces}~\term{lec}}}}}
```

In this example we can see that there contains 1 annotated `schemaText` but 3 `expressions`. Another reason why there may be more `expressions` than `schemaText` is because when annotating a specification with ZCGa, `declarations` also contain `expressions`. If we have the following example, again taken from the ProjectAlloc specification:

```
\text{\declaration{\set{studInterests}, \set{lecInterests}:
\expression{PERSON \pfun\iseq TOPIC}}}
```

The ZCGa text contains 1 annotation of `SchemaText`, 1 annotation of a `declaration`, 2 annotations of `sets` and 1 annotation of an `expression`. Since this is the case we expect to see more expressions than declarations in every specification, which is true according to table 12.3.

12.1.3 ZDRa Count

In this section we analyse the amount of ZDRa instances and relations are labeled for each of the specifications we translated. We give details of the amount of instances in table 12.4 and give details of the amount of relations in each specification in table 12.5.

Specification	**ZDRa Instances**									
	A	**SS**	**IS**	**CS**	**OS**	**TS**	**PRE**	**PO**	**O**	**SI**
Steamboiler	6	2	2	21	6	6	21	23	12	1
ProjectAlloc	0	1	1	5	11	0	11	6	22	1
VideoShop	0	1	1	3	10	0	13	4	20	1
TelephoneDirectory	0	1	1	4	5	5	8	5	10	1
ClubState	1	1	1	4	6	4	9	6	11	0
ZCGa	0	1	1	6	1	0	6	7	2	1
GenDB	0	1	1	4	2	0	6	5	4	1
Timetable	1	1	1	4	0	0	4	5	0	1
BirthdayBook	0	1	1	1	4	2	4	2	8	1
AutoPilot	0	2	0	1	1	0	1	1	2	0
ClubState2	1	2	1	3	0	0	3	4	0	2
Vending Machine	0	1	0	3	0	3	3	2	0	0
ModuleReg	0	1	0	2	0	0	2	2	0	1

Table 12.4: How many of each ZDRa instances exists in each specification.

From table 12.4 we can see that all specifications have either 1 or 2 statesS-chema's. For state base specification it should be the case that then specification has at least 1 state. Most state based specifications have stateInvariants that must be conformed to through all the changes of the specification. However this is not a must and some specification (even from our sample) do not have any stateInvariants.

All precondition must have a corresponding postcondition or output, therefore we can say:

Lemma 12.1.1. *precondition* $\longrightarrow$ *postcondition* $\vee$ *output*

The table supports this informatio as there are more combined postconditions and outputs then there are precondition. However not all postconditions and outputs need to have a precondition, they can be executed without one. Therefore the number of preconditions does not need to equal the total number of postcondition and outputs.

Specification	**ZDRa Relations**				
	initiaOf	**requires**	**allows**	**totalises**	**uses**
Steamboiler	2	28	21	24	92
ProjectAlloc	1	16	11	0	16
VideoShop	0	15	13	0	142
TelephoneDirectory	1	11	8	14	8
ClubState	1	12	9	14	12
ZCGa	1	9	6	0	7
GenDB	1	8	6	0	6
Timetable	1	6	4	0	6
BirthdayBook	1	7	4	6	5
AutoPilot	0	2	1	0	2
ClubState2	1	6	3	0	6
Vending Machine	0	2	0	2	8
ModuleReg	0	3	2	0	2

Table 12.5: How many of each ZDRa relations exists in each specification.

We can cross reference the table showing the amount of instances (table 12.4) with the table showing the relations (table 12.5). For example, the relation *initialOf* can only occur if the specification has an *initialSchema*. Not all specifictions have an *initialSchema* and therefore do not have an *initialOf* relation.

There is also an equal amount of *allows* relations as there is *preconditions*. As was written previously, all preconditions must have a corresponding output or post-condition, therefore the relation '*allows*' links each precondition to its corresponding postcondition or output. However, the vendingMachine specification is an exception to this as the preconditions are written as entire schema's. For example we have the following instance in the vending machine specification:

```
\draschema{PRE3}{
\begin{schema}{some\_stock}
stock: \nat
```

```
\where
stock > 0
\end{schema}}
```

This chunk of specification describes an entire schema as a precondition. The totalising schema then joints the precondition to their corresponding output or post-condition. The specification is written in this way as it is a personal choice of writing the specification formally. All other specifications in our sample set are written in the style where the precondition and corresponding output or postcondition are written inside the same schema environment.

Obviously, the relation '*totalises*' only occurs in specifications where totaliseSchema's are present. Therefore the '*totalise*' relation is not necessary in all specifications.

VideoShop specification is one of the largest specifications (in terms of lines of LaTeX) in our sample set however it has quite a small amount of relations

Complete Evaluation and discussion chapter

12.2 Case Studies

This section describes a few specification case studies in which we have used the ZMathLang tool kit to translate and prove formal specifications into the Isabelle automated theorem prover. The first case study present a formal specification only using terms, the second is a formal specification where both sets and terms are used and therefore the syntax used in Isabelle is more complex. The final case study we present is a partial translation of a specification which is not fully formalised but on it's way to becoming fully formal.

12.2.1 Case Study 1: A specification using only terms.

The following case study is based on the *Steamboiler* specification which has been translated and proved in Isabelle using the ZMathLang framework. This case study only uses variables which are terms.

12.2.2 Case Study 2: A specification using both terms and sets.

This case study based is on the *Project Allocation* specification which uses both terms and sets. The specification has been translated into Isabelle using the ZMathLang framework.

12.2.3 Case Study 3: A semi formal specification.

In this case study we present the *AutoPilot* specification. The specification is a semi formal specification and has been partially translated into Isabelle. The parts which have been translated are written formally and have been annotated accordingly. This gives an example of a specification which is written in natural language and is on it's way to being formalised.

We have taken the natural language specification for an autopilot system from [16] and started to formalise it.

The mode-control panel contains four buttons for selecting modes and three displays for dialing in or displaying values. The system supports the following four modes:

- attitude control wheel steering (att_cws)
- flight path angle selected (fpa_sel)
- altitude engage (alt_eng)
- calibrated air speed (cas_eng)

Only one of the first three modes can be engaged at any time. However, the cas_eng mode can be engaged at the same time as any of the other modes. The pilot engages a mode by pressing the corresponding button on the panel. One of the three modes, att_cws, fpa_sel, or alz_eng, should be engaged at all times. Engaging any of the first three modes will automatically cause the other two to be disengaged since only one of these three modes can be engaged at a time.

There are three displays on the panel: and altitude [ALT], flight path angle [FPA], and calibrated air speed [CAS]. The displays usually show the current values for the altitude, flight path angle, and air speed of the aircraft. However, the pilot can enter a new value into a display by dialing in the value using the knob next to the display. This is the target or "preselected" value that the pilot wishes the aircraft to attain. For example, if the pilot wishes to climb to 25,000 feet, he will dial 25,000 into the altitude display window and then press the alz_eng button to engage the altitude mode. Once the target value is achieved or the mode is disengaged, the display reverts to showing the "current" value.

If the pilot dials in an altitude that is more than 1,200 feet above the current altitude and then presses the alz_eng button, the altitude mode

Figure 12.1: An example of the original Autopilot specification.

The mode-control panel contains four buttons for selecting modes and three displays for dialing in or displaying values. The system supports the following four modes:

- attitude control wheel steering (att_cws)
- flight path angle selected (fpa_sel)
- altitude engage (alt_eng)
- calibrated air speed (cas_eng)

$events ::= press\_att\_cws \mid press\_cas\_eng \mid press\_alt\_eng \mid press\_fpa\_sel$

Only one of the first three modes can be engaged at any time. However, the cas_eng mode can be engaged at the same time as any of the other modes. The pilot engages a mode by pressing the corresponding button on the panel. One of the three modes, att_cws, fpa_sel, or alz_eng, should be engaged at all times. Engaging any of the first three modes will automatically cause the other two to be disengaged since only one of these three modes can be engaged at a time.

$mode\_status ::= off \mid engaged$

off_eng

$mode : mode\_status$

$mode = off \lor mode = engaged$

AutoPilot

Figure 12.2: An example of the Autopilot specification partially formalised.

We give the informal specification in figure 12.1 and one which we are beginning to formalised in figure 12.2. We have highlighted in red the parts which we have

formalised in figure 12.2. The formalised parts of the semi formal specification are taken from the text in the informal specification.

We then annotate the partial formal specification in ZCGa annotations and ZDRa annotations taken from chapters 4 and 5 respectively. Once annotated we can check the annotated document for ZCGa and ZDRa errors.

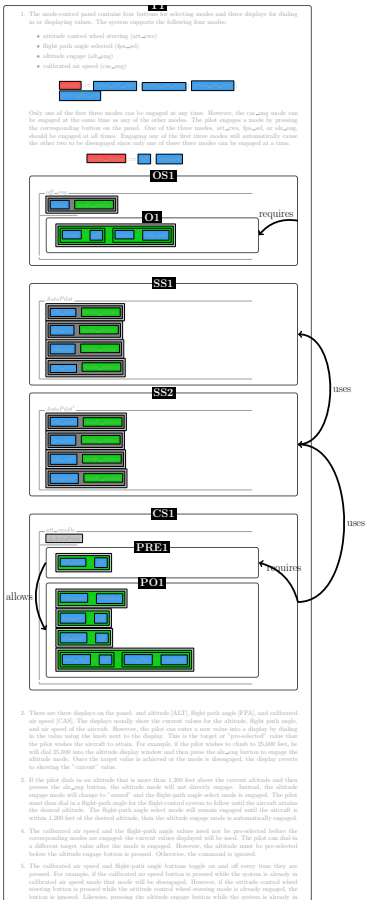


Figure 12.3: An example of the original Autopilot specification annotated in ZCGa and ZDRa.

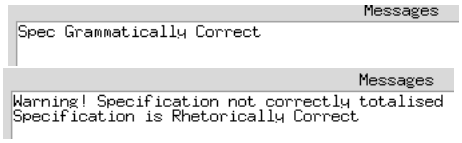


Figure 12.4: The outputting result when checking the autopilot specification with the ZCGa and ZDRa checkers.

Even though the specification is not fully formalised we can still annotate it with ZCGa and ZDRa and check for the correctness of the parts which have been annotated (shown in figures 12.3 and 12.4). When checking with ZDRa we have a warning message telling the user that the specification is not correctlty totalised. That is there is a precondition outstand with not postcondition counter part. This does not matter for now as we can still carry on with the tranlation.

When checking the specification for ZDRa, ZMathLang has also produced a

dependecy graph and goto graphs (shown in figures 12.5 and 12.6):

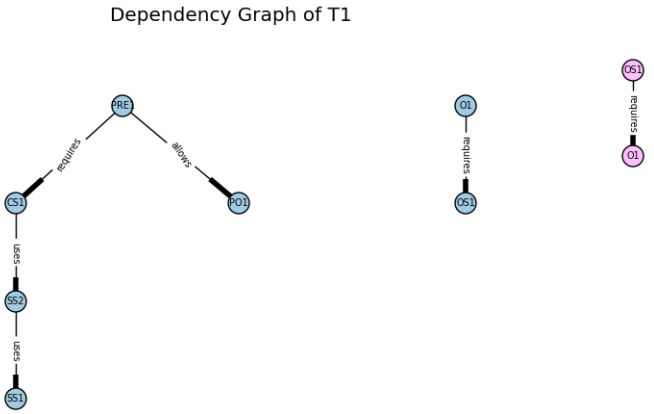


Figure 12.5: The dependecy graph produced for the autopilot specification.

Figure 12.6: The goto graph produced for the autopilot specification.

With the dependency graph (figure 12.5) we can say that $SS2$ uses $SS1$, $CS1$ uses $SS2$, $PRE1$ requires $CS1$ and allows $PO1$. Which makes up the main tree dependencies. $OS1$ and $O1$ are seperate as they do not have any relations which any parts of the main tree, the only dependency they have is on eachother where $O1$ requires $OS1$.

We can say that the dependecy graph *describes* the relation between instances and the goto graph (figure 12.6) *orders* the instances in a way as to parse through a theorem prover.

We can now generate a general proof skeleton for the Autopilot specification even though it is not fully formalised (shown in figure 12.7). We can clearly see that the arrow has changed direction for the $OS1$ and $O1$ relationship from the dependency graph. Again since these two instances are not dependency on any part of the main tree they are seperate. However in the dependency graph described the relation that $O1$ *requires* $OS1$ ($O1$ root and $OS1$ child) the goto graph flips this relationship as in a theorem prover we woud need $OS1$ to appear before $O1$ since $O1$ requires $OS1$ to exist. We can also say that $SS2$ uses $SS1$ therefore $SS2$ needs $SS1$ to exist for

itself to exist. Then $CS1$ uses $SS2$ therefore $CS1$ needs $SS2$ to exist for itself to exit. We can say that $PRE1$ requires $CS1$ and allows $PO1$. Thefore $PO1$ needs $PRE1$ to exist before it is allowed to exist itself.

```
stateSchema SS1

outputSchema OS1

output O1

stateSchema SS2

changeSchema CS1

precondition PRE1

postcondition PO1
```

Figure 12.7: Gpsa for the Autopilot specification.

Since the Autopilot specification has passed the ZCGa and ZDRa checks we can then generate a Gpsa for the specification using the goto graph produced in the previous stage. The way this is done is described in section 6.1. Note that even though there is a *changeSchema* instance, there are no *stateInvariants* in the specification (as yet). Therefore ZMathLang does not generate any lemma's to prove in this case since ZMathLang only checks for consistancy accross the specification and thus need state invariants to be present.

12.3 Analysing examples

12.4 Reflection and Discussion

12.5 Conclusion

Chapter 13

Conclusion and Future Work

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

13.1 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.

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

The first point is outlined in chapter 3 and described in detail in chapter 4. 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 4.

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 13.2.2.1. 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 13.2.2.4.

13.1.2 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 5. 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 9. 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 13.2.2.3.

13.1.3 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
- 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 5) show how the instances are related to eachother, the Gpsa (chapter 6) 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 7) uses the ZDRa instance names and creates a skeleton in Isabelle syntax. The halfbaked proof (chapter 7) 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 7.1 in chapter 7. The current syntax covers all the examples which are in the appendix and in [15]. 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 [72] 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 13.2.2.2.

13.1.4 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 3, 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 10. 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 11).

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 [59].

13.2 ZMathLang Current and Future Developments

13.2.1 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.

13.2.1.1 Other Developments

An extension to ZMathLang has started being developed by Fellar [29], [28] 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 [59], [58]. 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.

13.2.2 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.

13.2.2.1 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.

13.2.2.2 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.

13.2.2.3 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 specif-

cations 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.

13.2.2.4 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.

13.2.2.5 More than one system specification in one document

On occasions, some systems are made up of lots of smaller subsystems. Or sometimes one may want to design a system specification which are unrelated to eachother in one single document. Currently the ZCGa in the ZMathLang supports this. This is because the ZCGa checker first goes through the annotated specification and adds all correct categories into sets e.g correct terms[1] go into a python list called '`correct_terms`' and all correct sets go into a python list called '`correct_sets`' etc.

If the program reads the line `\specification{}`, which denotes a new specification, all these python lists a reset and the weak type checker starts again.

However the ZDRa does not have the ability to check multiple specifications e.g create many seperate dependency graphs or multiple Gpsa. If there are more than two specifications in a document and they both contain the same instance name (e.g `SS1`) then the ZDRa checker will regard this as the same instance and will ask to rename one of them. For future work, it may be ideal to do something similar. We

[1]By correct terms we mean terms which are labelled with ZCGa annotations such as `\term` and are weakly correctly typed.

can add the a function in the ZDRa program to reset all instances and relationships if it sees a new specification or '`\theory`'.

13.3 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.

Appendix A

Specifications in ZMathLang

The specifications translated using ZMathLang written here are the ones which a refered to in this thesis. The full range of examples of specifications can be found in [15].

A.1 Vending Machine

A.1.1 Raw Latex

```
\documentclass{article}
\usepackage{zed}
\begin{document}
\begin{zed}
price:\nat
\end{zed}

\begin{schema}{VMSTATE}
stock, takings: \nat
\end{schema}

\begin{schema}{VM\_operation}
\Delta VMSTATE \\
cash\_tendered?, cash\_refunded!: \nat \\
bars\_delivered! : \nat
\end{schema}

\begin{schema}{exact\_cash}
cash\_tendered?: \nat
\where
cash\_tendered? = price
\end{schema}

\begin{schema}{insufficient\_cash}
cash\_tendered? : \nat
\where
cash\_tendered? < price
\end{schema}

\begin{schema}{some\_stock}
stock: \nat
\where
stock > 0
\end{schema}

\begin{schema}{VM\_sale}
VM\_operation
\where
stock' = stock -1 \\
bars\_delivered! = 1 \\
cash\_refunded! = cash\_tendered? - price \\
takings' = takings + price
\end{schema}

\begin{schema}{VM\_nosale}
VM\_operation
\where
stock' = stock \\
bars\_delivered! = 0 \\
cash\_refunded! = cash\_tendered?\\
takings' = takings
\end{schema}

\begin{zed}
VM1 \defs exact\_cash \land some\_stock \land VM\_sale
\end{zed}

\begin{zed}
VM2 \defs insufficient\_cash \land VM\_nosale
\end{zed}

\begin{zed}
VM3 \defs VM1 \lor VM2
\end{zed}
\end{document}
```

A.1.2 Raw Latex output

$$
\begin{array}{|l}
\hline
\textit{VMSTATE} \\
\textit{stock}, \textit{takings} : \mathbb{N} \\
\hline
\end{array}
$$

$$
\begin{array}{|l}
\hline
\textit{VM\_operation} \\
\Delta \textit{VMSTATE} \\
\textit{cash\_tendered}?, \textit{cash\_refunded}! : \mathbb{N} \\
\textit{bars\_delivered}! : \mathbb{N} \\
\hline
\end{array}
$$

$$
\begin{array}{|l}
\hline
\textit{exact\_cash} \\
\textit{cash\_tendered}? : \mathbb{N} \\
\hline
\textit{cash\_tendered}? = \textit{price} \\
\hline
\end{array}
$$

$$
\begin{array}{|l}
\hline
\textit{insufficient\_cash} \\
\textit{cash\_tendered}? : \mathbb{N} \\
\hline
\textit{cash\_tendered}? < \textit{price} \\
\hline
\end{array}
$$

$$
\begin{array}{|l}
\hline
\textit{some\_stock} \\
\textit{stock} : \mathbb{N} \\
\hline
\textit{stock} > 0 \\
\hline
\end{array}
$$

$$
\begin{array}{|l}
\hline
\textit{VM\_sale} \\
\textit{VM\_operation} \\
\hline
\textit{stock}' = \textit{stock} - 1 \\
\textit{bars\_delivered}! = 1 \\
\textit{cash\_refunded}! = \textit{cash\_tendered}? - \textit{price} \\
\textit{takings}' = \textit{takings} + \textit{price} \\
\hline
\end{array}
$$

$$
\begin{array}{|l}
\hline
\textit{VM\_nosale} \\
\textit{VM\_operation} \\
\hline
\textit{stock}' = \textit{stock} \\
\textit{bars\_delivered}! = 0 \\
\textit{cash\_refunded}! = \textit{cash\_tendered}? \\
\textit{takings}' = \textit{takings} \\
\hline
\end{array}
$$

$$VM1 \mathrel{\widehat=} \textit{exact\_cash} \land \textit{some\_stock} \land \textit{VM\_sale}$$

$$VM2 \mathrel{\widehat=} \textit{insufficient\_cash} \land \textit{VM\_nosale}$$

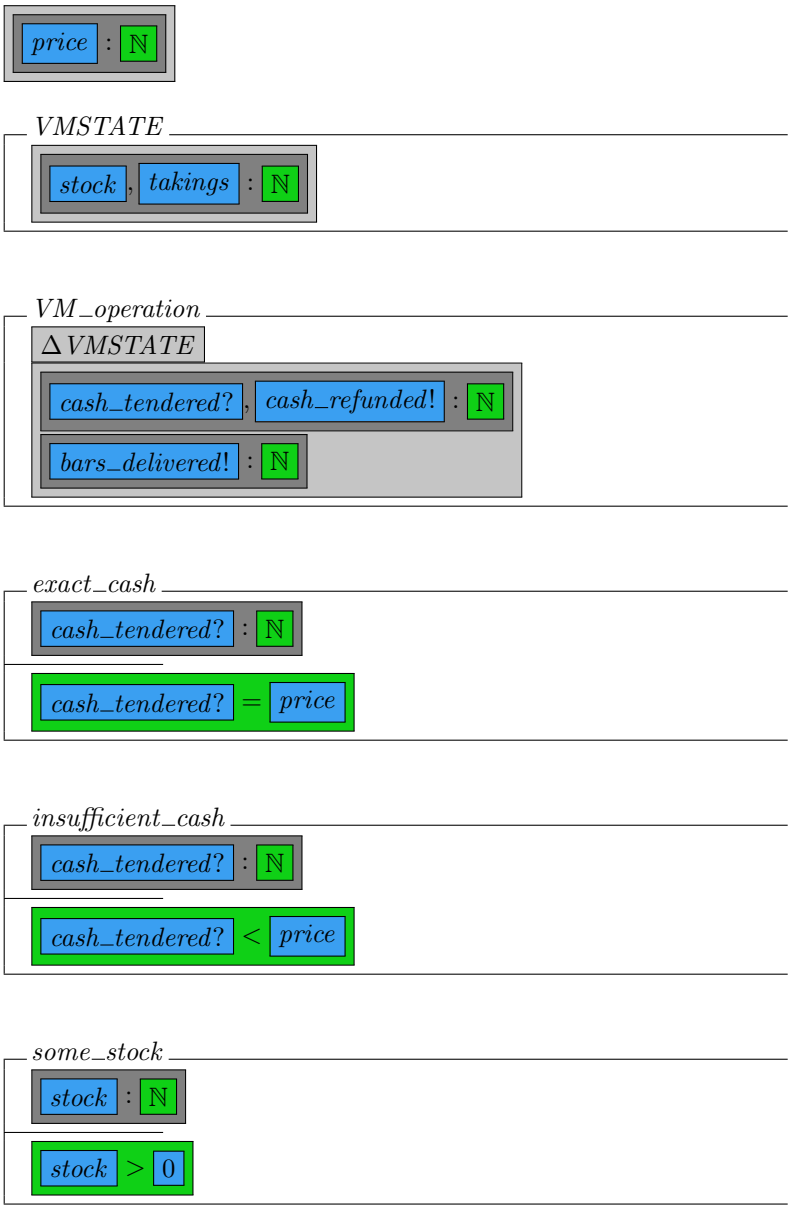
$$VM3 \mathrel{\widehat=} VM1 \lor VM2$$

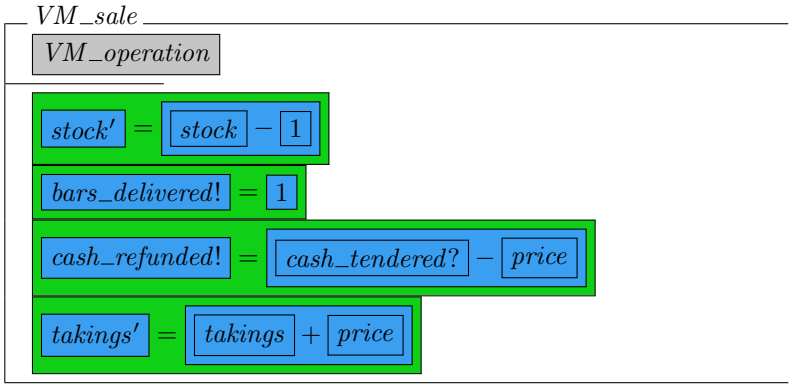
A.1.3 ZCGa Annotated Latex Code

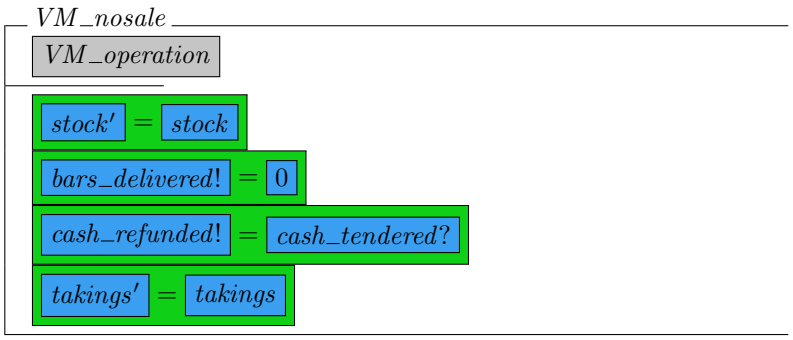
```
\documentclass{article}
\usepackage{zmathlang}
\begin{document}

\begin{zed}
\text{\declaration{\term{price}: \expression{\nat}}}
\end{zed}

\begin{schema}{VMSTATE}
\text{\declaration{\term{stock}, \term{takings}: \expression{\nat}}}
\end{schema}

\begin{schema}{VM\_operation}
\text{\Delta VMSTATE} \\
\text{\declaration{\term{cash\_tendered?},
\term{cash\_refunded!}: \expression{\nat}} \\
\declaration{\term{bars\_delivered!}: \expression{\nat}}}
\end{schema}

\begin{schema}{exact\_cash}
\declaration{\term{cash\_tendered?}: \expression{\nat}}
\where
\expression{\term{cash\_tendered?} = \term{price}}
\end{schema}

\begin{schema}{insufficient\_cash}
\declaration{\term{cash\_tendered?}: \expression{\nat}}
\where
\expression{\term{cash\_tendered?} < \term{price}}
\end{schema}

\begin{schema}{some\_stock}
\declaration{\term{stock}: \expression{\nat}}
\where
\expression{\term{stock} > \term{0}}
\end{schema}

\begin{schema}{VM\_sale}
\text{VM\_operation}
\where
\expression{\term{stock'} = \term{\term{stock} - \term{1}}} \\
\expression{\term{bars\_delivered!} = \term{1}} \\
\expression{\term{cash\_refunded!} =
\term{\term{cash\_tendered?} - \term{price}}} \\
\expression{\term{takings'} = \term{\term{takings} + \term{price}}}
\end{schema}
```

```
\begin{schema}{VM\_nosale}
\text{VM\_operation}
\where
\expression{\term{stock'} = \term{stock}} \\
\expression{\term{bars\_delivered!} = \term{0}} \\
\expression{\term{cash\_refunded!} = \term{cash\_tendered?}}\\
\expression{\term{takings'} = \term{takings}}
\end{schema}

\begin{zed}
VM1 \defs \text{\expression{\text{exact\_cash} \land
\text{some\_stock} \land \text{VM\_sale}}}
\end{zed}

\begin{zed}
VM2 \defs \text{\expression{\text{insufficient\_cash} \land
\text{VM\_nosale}}}
\end{zed}

\begin{zed}
VM3 \defs \text{\expression{\text{VM1} \lor \text{VM2}}}
\end{zed}

\end{document}
```

A.1.4 ZCGa output







$VM1 \mathrel{\widehat=} exact\_cash \land some\_stock \land VM\_sale$

$VM2 \mathrel{\widehat=} insufficient\_cash \land VM\_nosale$

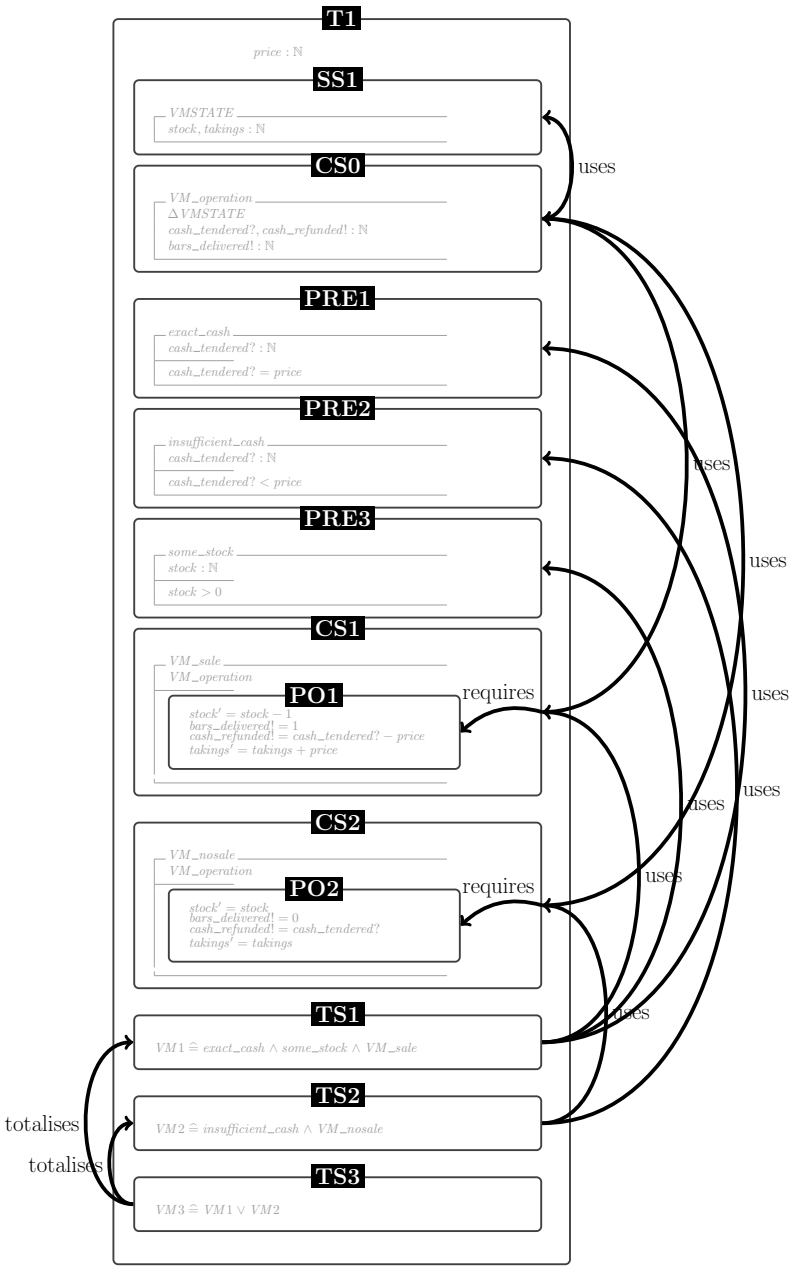
$VM3 \mathrel{\widehat=} VM1 \lor VM2$

A.1.5 ZDRa Annotated Latex Code

```
\documentclass{article}
\usepackage{zmathlang}
\begin{document}

\dratheory{T1}{0.5}{

\begin{zed}
price:\nat
\end{zed}

\draschema{SS1}{
\begin{schema}{VMSTATE}
stock, takings: \nat
\end{schema}}

\draschema{CS0}{
\begin{schema}{VM\_operation}
\Delta VMSTATE \\
cash\_tendered?, cash\_refunded!: \nat \\
bars\_delivered! : \nat
\end{schema}}

\uses{CS0}{SS1}

\draschema{PRE1}{
\begin{schema}{exact\_cash}
cash\_tendered?: \nat
\where
cash\_tendered? = price
\end{schema}}

\draschema{PRE2}{
\begin{schema}{insufficient\_cash}
cash\_tendered? : \nat
\where
cash\_tendered? < price
\end{schema}}

\draschema{PRE3}{
\begin{schema}{some\_stock}
stock: \nat
\where
stock > 0
\end{schema}}
```

```
\draschema{CS1}{
\begin{schema}{VM\_sale}
VM\_operation
\where
\draline{PO1}{stock' = stock -1 \\
bars\_delivered! = 1 \\
cash\_refunded! = cash\_tendered? - price \\
takings' = takings + price}
\end{schema}}

\uses{CS1}{CS0}
\requires{CS1}{PO1}

\draschema{CS2}{
\begin{schema}{VM\_nosale}
VM\_operation
\where
\draline{PO2}{stock' = stock \\
bars\_delivered! = 0 \\
cash\_refunded! = cash\_tendered?\\
takings' = takings}
\end{schema}}

\uses{CS2}{CS0}
\requires{CS2}{PO2}

\draschema{TS1}{
\begin{zed}
VM1 \defs exact\_cash \land some\_stock \land VM\_sale
\end{zed}}

\uses{TS1}{PRE1}
\uses{TS1}{PRE3}
\uses{TS1}{CS1}

\draschema{TS2}{
\begin{zed}
VM2 \defs insufficient\_cash \land VM\_nosale
\end{zed}}

\uses{TS2}{PRE2}
\uses{TS2}{CS2}
\draschema{TS3}{
\begin{zed}
VM3 \defs VM1 \lor VM2
\end{zed}}

\totalises{TS3}{TS1}
\totalises{TS3}{TS2}
}
\end{document}
```

A.1.6 ZDRa Output

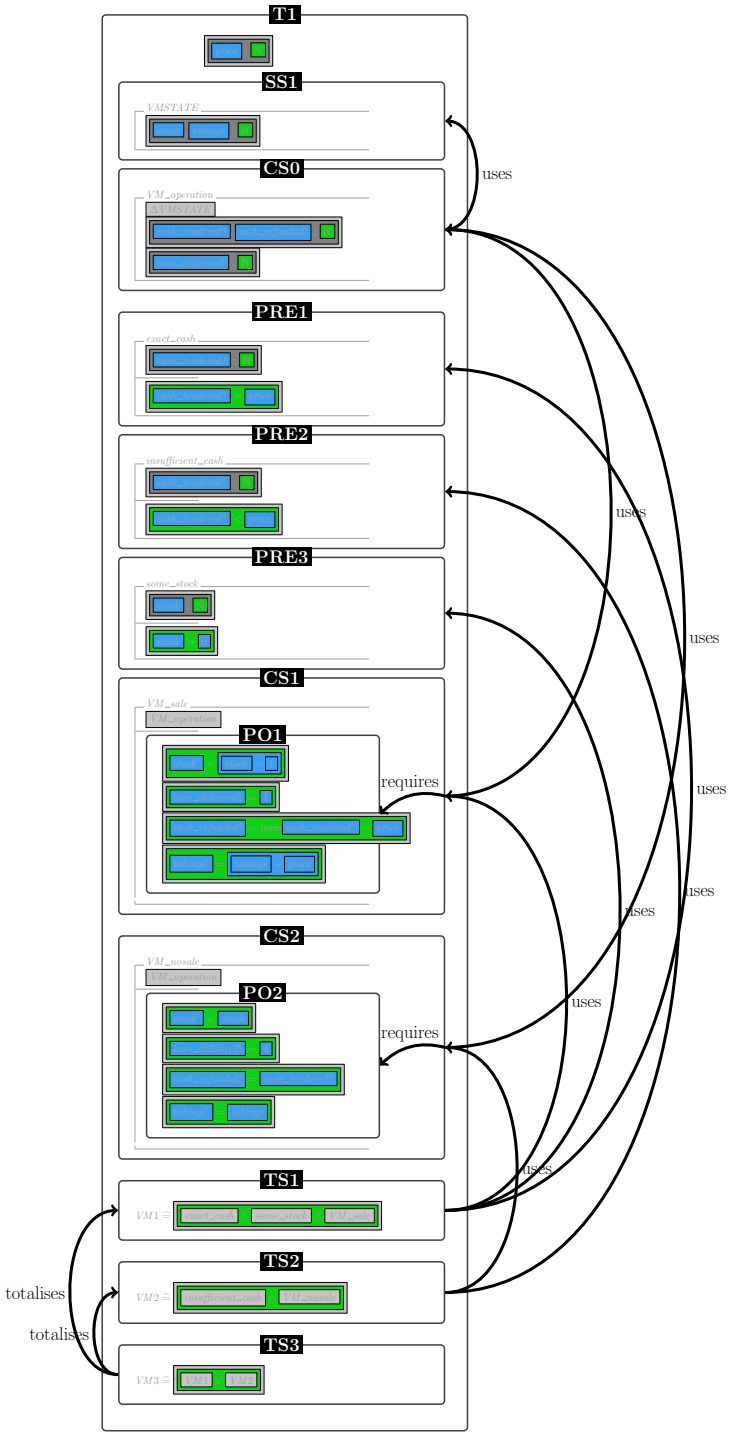


A.1.7 ZCGa and ZDRa Annotated Latex Code

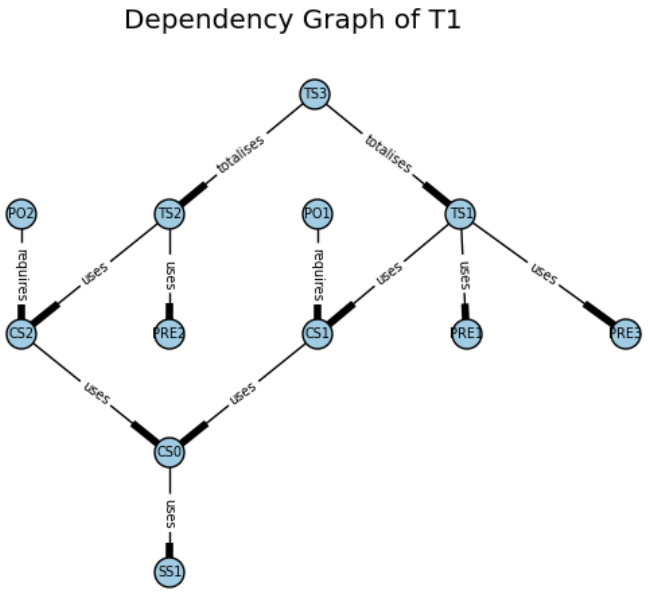
```
\documentclass{article}
\usepackage{zmathlang}
\begin{document}

\dratheory{T1}{0.5}{
\begin{zed}
\text{\declaration{\term{price}:\expression{\nat}}}
\end{zed}

\draschema{SS1}{
\begin{schema}{VMSTATE}
\text{\declaration{\term{stock}, \term{takings}: \expression{\nat}}}
\end{schema}}

\draschema{CS0}{
\begin{schema}{VM\_operation}
\text{\Delta VMSTATE} \\
\text{\declaration{\term{cash\_tendered?}, \term{cash\_refunded!}: \expression{\nat}}} \\
\text{\declaration{\term{bars\_delivered!}: \expression{\nat}}}
\end{schema}}

\uses{CS0}{SS1}

\draschema{PRE1}{
\begin{schema}{exact\_cash}
\text{\declaration{\term{cash\_tendered?}: \expression{\nat}}}
\where
\text{\expression{\term{cash\_tendered?} = \term{price}}}
\end{schema}}

\draschema{PRE2}{
\begin{schema}{insufficient\_cash}
\text{\declaration{\term{cash\_tendered?}: \expression{\nat}}}
\where
\text{\expression{\term{cash\_tendered?} < \term{price}}}
\end{schema}}

\draschema{PRE3}{
\begin{schema}{some\_stock}
\text{\declaration{\term{stock}: \expression{\nat}}}
\where
\text{\expression{\term{stock} > \term{0}}}
\end{schema}}
```

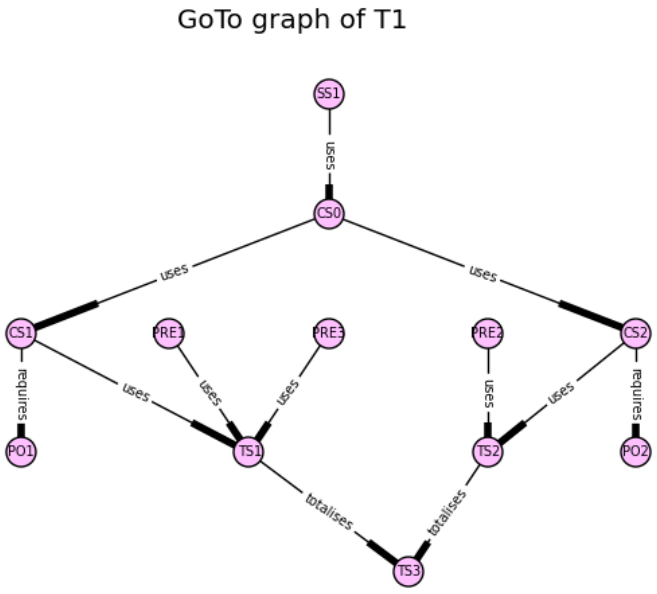
```
\draschema{CS1}{
\begin{schema}{VM\_sale}
\text{VM\_operation}
\where
\draline{PO1}{
\text{\expression{\term{stock'} = \term{\term{stock} - \term{1}}}} \\
\text{\expression{\term{bars\_delivered!} = \term{1}}} \\
\text{\expression{\term{cash\_refunded!} = term{\term{cash\_tendered?} - \term{price}}}} \\
\text{\expression{\term{takings'} = \term{\term{takings} + \term{price}}}}}
\end{schema}}

\uses{CS1}{CS0}
\requires{CS1}{PO1}

\draschema{CS2}{
\begin{schema}{VM\_nosale}
\text{VM\_operation}
\where
\draline{PO2}{\text{\expression{\term{stock'} = \term{stock}}} \\
\text{\expression{\term{bars\_delivered!} = \term{0}}} \\
\text{\expression{\term{cash\_refunded!} = \term{cash\_tendered?}}}\\
\text{\expression{\term{takings'} = \term{takings}}}}
\end{schema}}

\uses{CS2}{CS0}
\requires{CS2}{PO2}

\draschema{TS1}{
\begin{zed}
VM1 \defs \text{\expression{\text{exact\_cash} \land
\text{some\_stock} \land \text{VM\_sale}}}
\end{zed}}

\uses{TS1}{PRE1}
\uses{TS1}{PRE3}
\uses{TS1}{CS1}

\draschema{TS2}{
\begin{zed}
VM2 \defs \text{\expression{\text{insufficient\_cash}
\land\text{VM\_nosale}}}
\end{zed}}

\uses{TS2}{PRE2}
\uses{TS2}{CS2}

\draschema{TS3}{
\begin{zed}
VM3 \defs \text{\expression{\text{VM1} \lor \text{VM2}}}
\end{zed}}

\totalises{TS3}{TS1}
\totalises{TS3}{TS2}
}
\end{document}
```

A.1.8 ZCGa and ZDRa Output



A.1.9 Dependency and Goto Graphs





A.1.10 General Proof Skeleton

```
stateSchema SS1
```

```
precondition PRE1

precondition PRE2

changeSchema CS0

precondition PRE3

changeSchema CS2

postcondition PO2

totaliseSchema TS2

changeSchema CS1

postcondition PO1

totaliseSchema TS1

totaliseSchema TS3
```

A.1.11 Isabelle Proof Skeleton

```
theory isaSkeleton_vendingmachine
imports
Main
begin

record SS1 =
(*DECLARATIONS*)

locale zmathlang_vm =
fixes (*GLOBAL DECLARATIONS*)
begin

definition PRE1 ::
 "(*PRE1_TYPES*) => bool"
where
"PRE1 (*PRE1_VARIABLES*) == (*PRECONDITION*) "

definition PRE2 ::
 "(*PRE2_TYPES*) => bool"
where
"PRE2 (*PRE2_VARIABLES*) == (*PRECONDITION*) "

definition CS0 ::
"(*CS0_TYPES*) => bool"
where
"CS0 (*CS0_VARIABLES*) == True"

definition PRE3 ::
 "(*PRE3_TYPES*) => bool"
where
"PRE3 (*PRE3_VARIABLES*) == (*PRECONDITION*) "
```

```
definition CS0 ::
"(*CS0_TYPES*) => bool"
where
"CS0 (*CS0_VARIABLES*) == True"

definition PRE3 ::
 "(*PRE3_TYPES*) => bool"
where
"PRE3 (*PRE3_VARIABLES*) == (*PRECONDITION*) "

definition CS2 ::
 "(*CS2_TYPES*) => bool"
where
"CS2 (*CS2_VARIABLES*) == (PO2)"

lemma TS2:
"(*TS2_EXPRESSION*)"
sorry

definition CS1 ::
 "(*CS1_TYPES*) => bool"
where
"CS1 (*CS1_VARIABLES*) == (PO1)"

lemma TS1:
"(*TS1_EXPRESSION*)"
sorry

lemma TS3:
"(*TS3_EXPRESSION*)"
sorry

end
end
```

A.1.12 Isabelle Filled In

```
theory 5
imports
Main

begin

record VMSTATE =
STOCK :: nat
TAKINGS :: nat

locale zmathlang_vm =
fixes price :: nat
begin

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

definition exact_cash ::
 "nat  ⇒ bool"
where
"exact_cash cash_tendered  ==
 cash_tendered = price"

 definition VM_operation ::
"VMSTATE => VMSTATE => nat => nat => nat => bool"
where
" VM_operation vmstate vmstate' cash_tendered
cash_refunded bars_delivered == True"

definition some_stock ::
 "nat => bool"
where
" some_stock stock ==  stock > 0  "
```

```
definition VM_nosale ::
 "nat => nat => nat => nat => nat => nat => nat => bool"
where
" VM_nosale stock takings stock' takings'
cash_tendered cash_refunded bars_delivered == ((stock' = stock)
∧ (bars_delivered = 0)
∧ (cash_refunded = cash_tendered)
∧ (takings' = takings))"

definition VM2 ::
 "nat  => nat => nat => nat => nat => nat => nat => bool"
where
" VM2  cash_tendered stock takings stock' takings'
cash_refunded bars_delivered ==
(insufficient_cash  cash_tendered)
 ∧ (VM_nosale stock takings stock' takings' cash_tendered
 cash_refunded bars_delivered)"

definition VM_sale :: " nat => nat => nat ⇒nat => nat =>
nat => nat => bool"
where
" VM_sale  stock takings stock' takings' cash_tendered
cash_refunded bars_delivered ==
(stock' = stock - 1)
∧ (bars_delivered = 1)
∧ (cash_refunded = cash_tendered - price)
∧ (takings' = takings + price) "
```

```
definition VM_sale :: " nat => nat => nat ⇒nat => nat =>
nat => nat => bool"
where
" VM_sale  stock takings stock' takings' cash_tendered
cash_refunded bars_delivered ==
(stock' = stock - 1)
∧ (bars_delivered = 1)
∧ (cash_refunded = cash_tendered - price)
∧ (takings' = takings + price) "

definition VM1 ::
 "nat ⇒ nat  => nat => nat => nat => nat  => nat => bool"
where
" VM1  cash_tendered stock takings stock' takings'
cash_refunded bars_delivered ==
(exact_cash cash_tendered )
∧ (some_stock stock)
∧ (VM_sale  stock takings stock' takings'
 cash_tendered cash_refunded bars_delivered)"

definition VM3 ::
 "nat  => nat => nat => nat => nat => nat => nat => bool"
where
" VM3  cash_tendered stock takings stock' takings'
cash_refunded bars_delivered =
((VM1  cash_tendered stock takings stock' takings'
cash_refunded bars_delivered)
 ∨ (VM2  cash_tendered stock takings stock' takings'
cash_refunded bars_delivered)) "

 end
 end
```

A.1.13 Full Proof in Isabelle

```
theory 6
imports
Main

begin

record VMSTATE =
STOCK :: nat
TAKINGS :: nat

locale zmathlang_vm =
fixes price :: nat
begin

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

definition exact_cash ::
 "nat  ⇒ bool"
where
"exact_cash cash_tendered  ==
 cash_tendered = price"

 definition VM_operation ::
"VMSTATE => VMSTATE => nat => nat => nat => bool"
where
" VM_operation vmstate vmstate' cash_tendered
cash_refunded bars_delivered == True"

definition some_stock ::
 "nat => bool"
where
" some_stock stock ==  stock > 0  "
```

```
definition VM_nosale ::
 "nat => nat => nat => nat => nat => nat => nat => bool"
where
" VM_nosale stock takings stock' takings'
cash_tendered cash_refunded bars_delivered == ((stock' = stock)
∧ (bars_delivered = 0)
∧ (cash_refunded = cash_tendered)
∧ (takings' = takings))"

definition VM2 ::
 "nat  => nat => nat => nat => nat => nat => nat => bool"
where
" VM2  cash_tendered stock takings stock' takings'
cash_refunded bars_delivered ==
(insufficient_cash  cash_tendered)
 ∧ (VM_nosale stock takings stock' takings' cash_tendered
 cash_refunded bars_delivered)"

definition VM_sale :: " nat => nat => nat ⇒nat => nat =>
nat => nat => bool"
where
" VM_sale  stock takings stock' takings' cash_tendered
cash_refunded bars_delivered ==
(stock' = stock - 1)
∧ (bars_delivered = 1)
∧ (cash_refunded = cash_tendered - price)
∧ (takings' = takings + price) "
```

```
definition VM1 ::
 "nat ⇒ nat  => nat => nat => nat => nat  => nat => bool"
where
" VM1  cash_tendered stock takings stock' takings'
cash_refunded bars_delivered ==
(exact_cash cash_tendered )
∧ (some_stock stock)
∧ (VM_sale  stock takings stock' takings'
 cash_tendered cash_refunded bars_delivered)"

definition VM3 ::
 "nat  => nat => nat => nat => nat => nat => nat => bool"
where
" VM3  cash_tendered stock takings stock' takings'
cash_refunded bars_delivered =
((VM1  cash_tendered stock takings stock' takings'
cash_refunded bars_delivered)
 ∨ (VM2  cash_tendered stock takings stock' takings'
 cash_refunded bars_delivered)) "

lemma pre_VM1:
"(∃ stock' takings' cash_refunded bars_delivered.
 VM1 cash_tendered stock takings stock' takings'
 cash_refunded bars_delivered)
 ⟷ (0 < stock) ∧
 (cash_tendered = price) ∧ (0 ≤ takings)"
 apply (unfold VM1_def exact_cash_def
 some_stock_def VM_sale_def)
 apply auto
 done
```

```
  lemma pre_VM2:
"(∃ stock' takings' cash_refunded bars_delivered.
 VM2 cash_tendered stock takings stock' takings'
 cash_refunded bars_delivered)
 ⟷ (cash_tendered < price) ∧
(cash_tendered ≥ 0) ∧ (stock ≥ 0) ∧ (takings ≥ 0)"
apply (unfold VM2_def insufficient_cash_def VM_nosale_def )
apply auto
done

  lemma pre_VM3:
"(∃ stock' takings' cash_refunded bars_delivered.
 VM3 cash_tendered stock takings stock'
 takings' cash_refunded bars_delivered)
 ⟷ (0 < stock ∧
 cash_tendered = price ∧ 0 ≤ takings) ∨ (cash_tendered < price)
∧ (0 ≤ cash_tendered)
∧ (0 ≤ stock)
∧ (0 ≤ takings)"
apply (unfold VM3_def VM2_def VM1_def
some_stock_def exact_cash_def VM_sale_def
  VM_nosale_def insufficient_cash_def)
apply auto
done

lemma cash_lemma: "¬ (insufficient_cash
cash_tendered ∧ exact_cash cash_tendered)"
apply (unfold insufficient_cash_def exact_cash_def)
apply auto
done
```

```
lemma VM3_refines_VM1:
"(∃ stock' takings' cash_refunded bars_delivered.
((VM1 cash_tendered stock takings stock' takings' cash_refunded
bars_delivered)
⟶
(VM3 cash_tendered stock takings stock' takings' cash_refunded
bars_delivered))
∧
(((VM1 cash_tendered stock takings stock' takings' cash_refunded
bars_delivered)
∧
(VM3 cash_tendered stock takings stock' takings' cash_refunded
bars_delivered))
⟶
(VM1 cash_tendered stock takings stock' takings' cash_refunded
bars_delivered)))"
apply (unfold VM3_def VM1_def VM_sale_def
 exact_cash_def some_stock_def)
apply auto
done

lemma VM3_ok:
"(∃ stock' takings cash_refunded bars_delivered.
(VM3 cash_tendered stock takings stock' takings' cash_refunded
bars_delivered)
⟶
((takings' - takings) ≥ price * (stock - stock' )))"
apply (unfold VM3_def VM1_def VM2_def exact_cash_def some_stock_def
  VM_sale_def VM_nosale_def insufficient_cash_def)
apply auto
done

end
end
```

A.2 BirthdayBook

A.2.1 Raw Latex

```
\documentclass{article}
\usepackage{zmathlang}

\begin{document}

\begin{zed}
   [NAME, ~ DATE]
\end{zed}

\begin{schema}{BirthdayBook}
known: \power NAME \\
birthday: NAME \pfun DATE
\where
known=\dom birthday
\end{schema}

\begin{schema}{InitBirthdayBook}
BirthdayBook~'
\where
known' = \{ \}
\end{schema}

\begin{schema}{AddBirthday}
\Delta BirthdayBook \\
name?: NAME \\
date?: DATE
\where
name? \notin known\\
birthday' = birthday \cup \{name? \mapsto date?\}
\end{schema}

\begin{schema}{FindBirthday}
\Xi BirthdayBook \\
name?: NAME \\
date!: DATE
\where
name? \in known\\
date! = birthday(name?)
\end{schema}

\begin{zed}
REPORT ::= ok | already\_known | not\_known
\end{zed}

\begin{schema}{Success}
result!: REPORT
\where
result! = ok
\end{schema}

\begin{schema}{AlreadyKnown}
\Xi BirthdayBook \\
name?: NAME \\
result!: REPORT
\where
name? \in known \\
result! = already\_known
\end{schema}

\begin{schema}{NotKnown}
\Xi BirthdayBook \\
name?: NAME \\
result!: REPORT
\where
name? \notin known \\
result! = not\_known
\end{schema}

\begin{zed}
RAddBirthday ==\\ (AddBirthday \land Success)\\
  \lor AlreadyKnown \\
RFindBirthday ==\\ (FindBirthday \land Success)
 \lor NotKnown \\
\end{zed}

\end{document}
```

A.2.2 Raw Latex ouput

$[NAME,\ DATE]$

BirthdayBook

$$known : \mathbb{P}\ NAME$$
$$birthday : NAME \pfun DATE$$
$$known = \mathrm{dom}\ birthday$$

InitBirthdayBook

$$BirthdayBook'$$
$$known' = \{\}$$

AddBirthday

$$\Delta BirthdayBook$$
$$name? : NAME$$
$$date? : DATE$$
$$name? \notin known$$
$$birthday' = birthday \cup \{name? \mapsto date?\}$$

FindBirthday

$$\Xi BirthdayBook$$
$$name? : NAME$$
$$date! : DATE$$
$$name? \in known$$
$$date! = birthday(name?)$$

$REPORT ::= ok \mid already\_known \mid not\_known$

Success

$$result! : REPORT$$
$$result! = ok$$

AlreadyKnown

$$\Xi BirthdayBook$$
$$name? : NAME$$
$$result! : REPORT$$
$$name? \in known$$
$$result! = already\_known$$

NotKnown

$\Xi BirthdayBook$
$name? : NAME$
$result! : REPORT$

$name? \notin known$
$result! = not\_known$

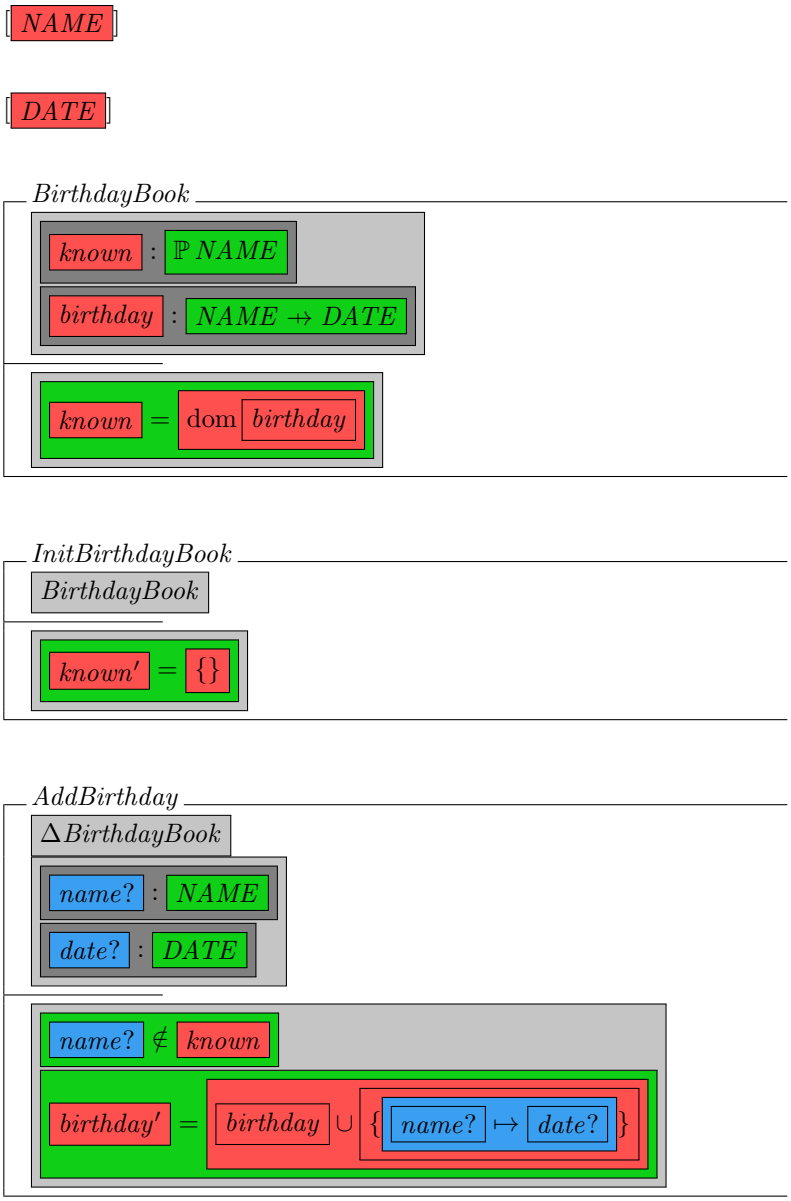
$RAddBirthday ==$
$(AddBirthday \land Success)$
$\lor AlreadyKnown$
$RFindBirthday ==$
$(FindBirthday \land Success) \lor NotKnown$

A.2.3 ZCGa Annotated Latex Code

```
\documentclass{article}
\usepackage{zmathlang}

\begin{document}

\begin{zed}
[\set{NAME}]
\end{zed}

\begin{zed}
[\set{DATE}]
\end{zed}

\begin{schema}{BirthdayBook}
\text{\declaration{\set{known}: \expression{\power NAME}} \\
\declaration{\set{birthday}: \expression{NAME \pfun DATE}}}
\where
\text{\expression{\set{known}=\set{\dom \set{birthday}}}}
\end{schema}

\begin{schema}{InitBirthdayBook}
\text{BirthdayBook}
\where
\text{\expression{\set{known'} = \set{\{ \}}}}
\end{schema}

\begin{schema}{AddBirthday}
\text{\Delta BirthdayBook} \\
\text{\declaration{\term{name?}: \expression{NAME}} \\
\declaration{\term{date?}: \expression{DATE}}}
\where
\text{\expression{\term{name?} \notin \set{known}}\\
\expression{\set{birthday'} = \set{\set{birthday}
\cup \set{\{\term{\term{name?} \mapsto \term{date?}}\}}}}}
\end{schema}

\begin{schema}{FindBirthday}
\text{\Xi BirthdayBook} \\
\text{\declaration{\term{name?}: \expression{NAME}} \\
\declaration{\term{date!}: \expression{DATE}} }
\where
\text{\expression{\term{name?} \in \set{known}}\\
\expression{\term{date!} = \term{\set{birthday}(\term{name?})}}}
\end{schema}

\begin{zed}
\set{REPORT} ::= \term{ok} | \term{already\_known}
| \term{not\_known}
\end{zed}

\begin{schema}{Success}
\text{\declaration{\term{result!}: \expression{REPORT}}}
\where
\text{\expression{\term{result!} = \term{ok}}}
\end{schema}
```

```
\begin{schema}{AlreadyKnown}
\text{\Xi BirthdayBook} \\
\text{\declaration{\term{name?}: \expression{NAME}} \\
\declaration{\term{result!}: \expression{REPORT}}}
\where
\text{\expression{\term{name?} \in \set{known}} \\
\expression{\term{result!} = \term{already\_known}}}
\end{schema}

\begin{schema}{NotKnown}
\text{\Xi BirthdayBook} \\
\text{\declaration{\term{name?}: \expression{NAME}} \\
\declaration{\term{result!}: \expression{REPORT}}}
\where
\text{\expression{\term{name?} \notin \set{known}} \\
\expression{\term{result!} = \term{not\_known}}}
\end{schema}

\begin{zed}
RAddBirthday ==\\
\text{\expression{(\text{AddBirthday} \land
\text{Success})\\  \lor \text{AlreadyKnown}}} \\
RFindBirthday ==\\
\text{\expression{(\text{FindBirthday}
\land \text{Success}) \lor \text{NotKnown}}} \\
\end{zed}

\end{document}
```

A.2.4 ZCGa output



FindBirthday

$\Xi BirthdayBook$

$name? : NAME$

$date! : DATE$

$name? \in known$

$date! = birthday(name?)$

$REPORT ::= ok \mid already\_known \mid not\_known$

Success

$result! : REPORT$

$result! = ok$

AlreadyKnown

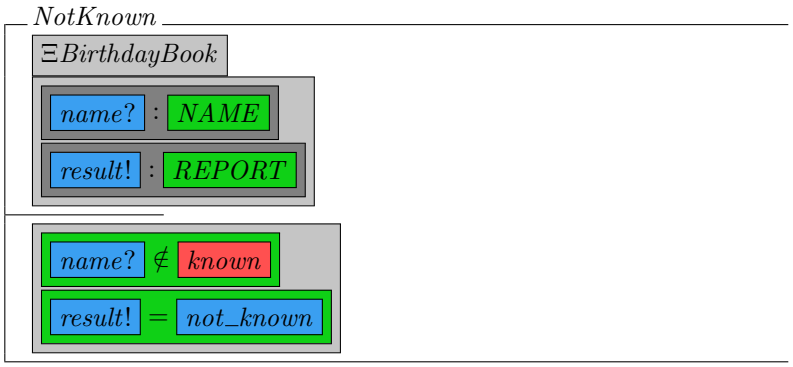
$\Xi BirthdayBook$

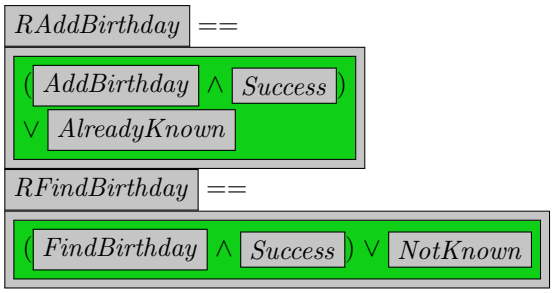
$name? : NAME$

$result! : REPORT$

$name? \in known$

$result! = already\_known$


NotKnown
ΞBirthdayBook
name? : NAME
result! : REPORT
name? ∉ known
result! = not_known

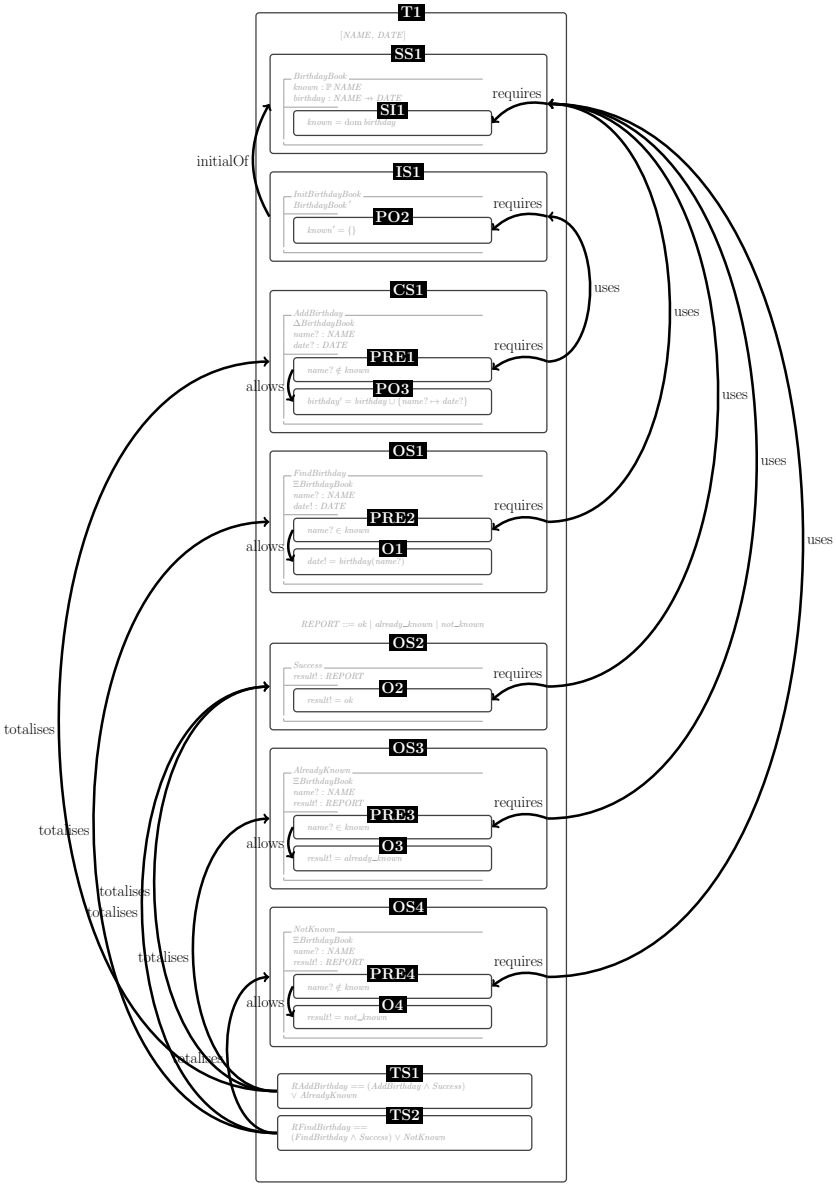

RAddBirthday ==
(AddBirthday ∧ Success)
∨ AlreadyKnown
RFindBirthday ==
(FindBirthday ∧ Success) ∨ NotKnown

A.2.5 ZDRa Annotated Latex Code

```
\documentclass{article}
\usepackage{zmathlang}

\begin{document}

\dratheory{T1}{0.34}{

\begin{zed}
[NAME,  DATE]
\end{zed}

\draschema{SS1}{
\begin{schema}{BirthdayBook}
known: \power NAME \\
birthday: NAME \pfun DATE
\where
\draline{SI1}{known=\dom birthday}
\end{schema}}

\requires{SS1}{SI1}

\draschema{IS1}{
\begin{schema}{InitBirthdayBook}
BirthdayBook~'
\where
\draline{PO2}{known' = \{ \}}
\end{schema}}

\requires{IS1}{PO2}
\initialof{IS1}{SS1}

\draschema{CS1}{
\begin{schema}{AddBirthday}
\Delta BirthdayBook \\
name?: NAME \\
date?: DATE
\where
\draline{PRE1}{name? \notin known}\\
\draline{PO3}{birthday' = birthday \cup
\{name? \mapsto date?\}}
\end{schema}}

\uses{CS1}{IS1}
\requires{CS1}{PRE1}
\allows{PRE1}{PO3}
```

```
\draschema{OS1}{
\begin{schema}{FindBirthday}
\Xi BirthdayBook \\
name?: NAME \\
date!: DATE
\where
\draline{PRE2}{name? \in known}\\
\draline{O1}{date! = birthday(name?)}
\end{schema}}

\allows{PRE2}{O1}
\uses{OS1}{SS1}
\requires{OS1}{PRE2}

\begin{zed}
REPORT ::= ok | already\_known | not\_known
\end{zed}

\draschema{OS2}{
\begin{schema}{Success}
result!: REPORT
\where
\draline{O2}{result! = ok}
\end{schema}}

\requires{OS2}{O2}
\uses{OS2}{SS1}

\draschema{OS3}{
\begin{schema}{AlreadyKnown}
\Xi BirthdayBook \\
name?: NAME \\
result!: REPORT
\where
\draline{PRE3}{name? \in known} \\
\draline{O3}{result! = already\_known}
\end{schema}}

\requires{OS3}{PRE3}
\allows{PRE3}{O3}
\uses{OS3}{SS1}

\draschema{OS4}{
\begin{schema}{NotKnown}
\Xi BirthdayBook \\
name?: NAME \\
result!: REPORT
\where
\draline{PRE4}{name? \notin known} \\
\draline{O4}{result! = not\_known}
\end{schema}}

\requires{OS4}{PRE4}
\allows{PRE4}{O4}
\uses{OS4}{SS1}

\begin{zed}
\draline{TS1}{RAddBirthday == (AddBirthday \land Success)
  \lor AlreadyKnown} \\
   \draline{TS2}{RFindBirthday ==\\ (FindBirthday \land Success)
\lor NotKnown} \\
\end{zed}

\totalises{TS1}{CS1}
\totalises{TS1}{OS2}
\totalises{TS1}{OS3}
\totalises{TS2}{OS1}
\totalises{TS2}{OS2}
\totalises{TS2}{OS4}
}

\end{document}
```

A.2.6 ZDRa Output

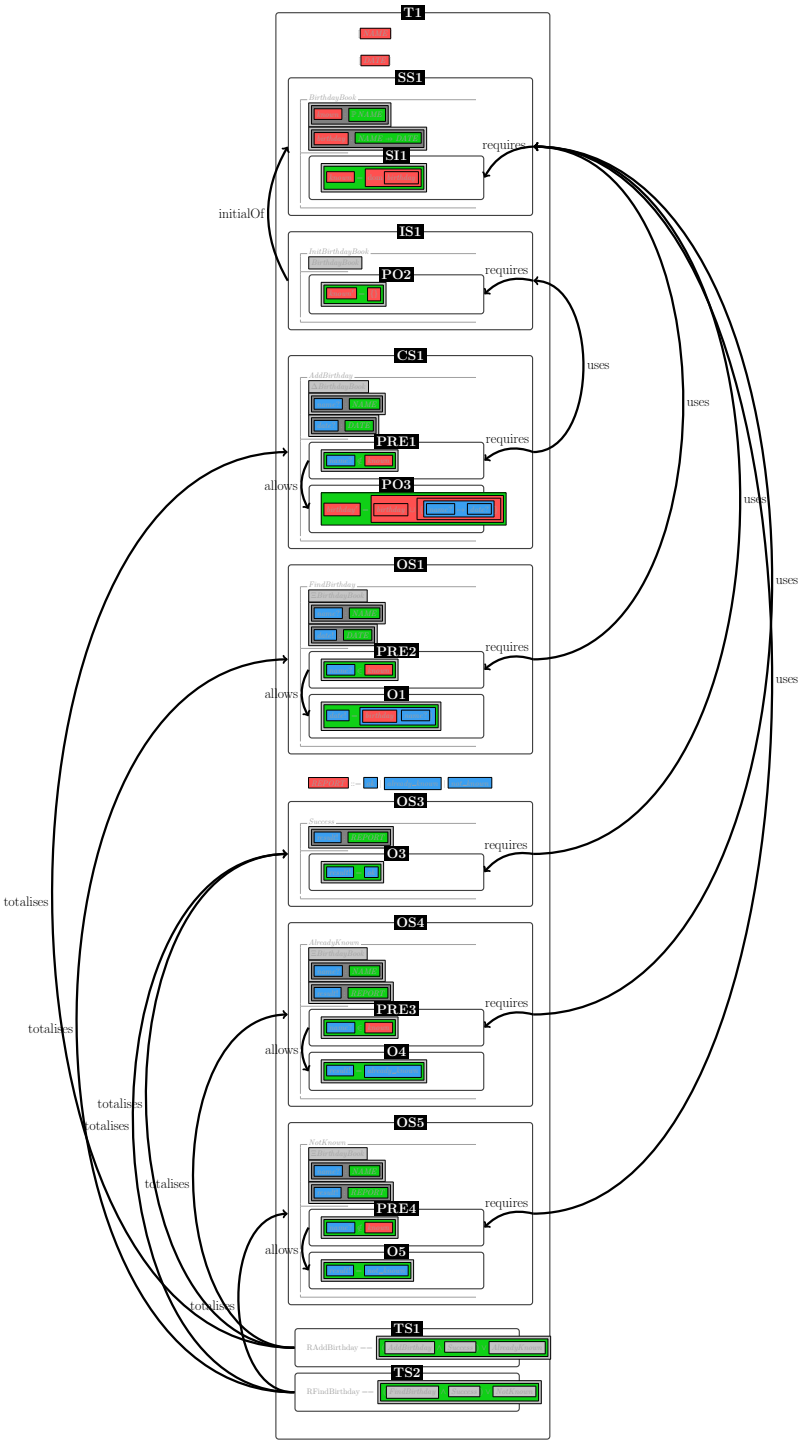


A.2.7 ZCGa and ZDRa Annotated Latex Code

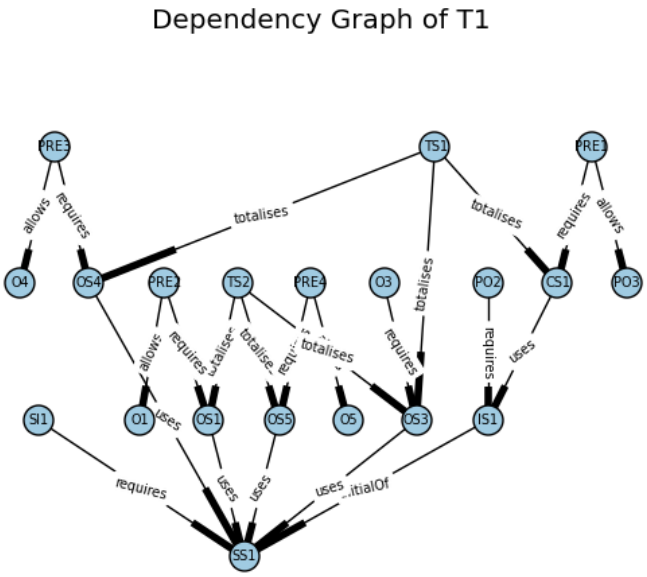
```
\documentclass{article}
\usepackage{zmathlang}

\begin{document}

\dratheory{T1}{0.3}{

\begin{zed}
[\set{NAME}]
\end{zed}

\begin{zed}
[\set{DATE}]
\end{zed}

\draschema{SS1}{
\begin{schema}{BirthdayBook}
\text{\declaration{\set{known}:\expression{\power NAME}}} \\
\text{\declaration{\set{birthday}:\expression{NAME \pfun DATE}}}
\where
\draline{SI1}{\text{\expression{\set{known}=\set{\dom \set{birthday}}}}}
\end{schema}}

\requires{SS1}{SI1}

\draschema{IS1}{
\begin{schema}{InitBirthdayBook}
\text{BirthdayBook}
\where
\draline{PO2}{\text{\expression{\set{known'} = \set{\{ \}}}}}
\end{schema}}

\requires{IS1}{PO2}

\initialof{IS1}{SS1}

\draschema{CS1}{
\begin{schema}{AddBirthday}
\text{\Delta BirthdayBook} \\
\text{\declaration{\term{name?}: \expression{NAME}}} \\
\text{\declaration{\term{date?}: \expression{DATE}}}
\where
\draline{PRE1}{\text{\expression{\term{name?} \notin \set{known}}}}\\
\draline{PO3}{\text{\expression{\set{birthday'} = \set{\set{birthday}
\cup \set{\{\term{\term{name?} \mapsto \term{date?}}\}}}}}}
\end{schema}}

\uses{CS1}{IS1}
\requires{CS1}{PRE1}
\allows{PRE1}{PO3}
```

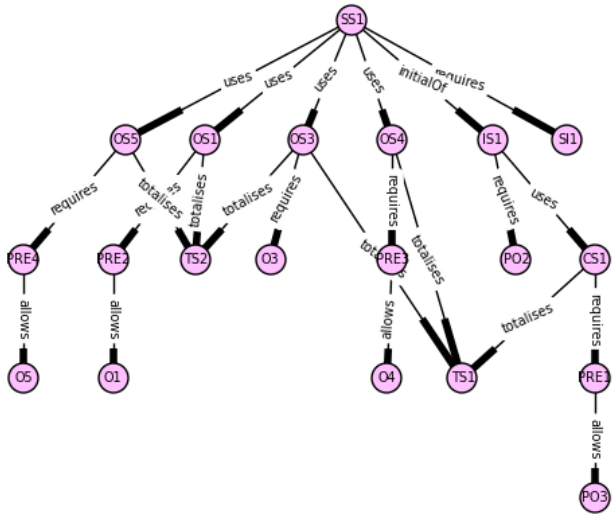
```
\draschema{OS1}{
\begin{schema}{FindBirthday}
\text{\Xi BirthdayBook} \\
\text{\declaration{\term{name?}: \expression{NAME}}} \\
\text{\declaration{\term{date!}: \expression{DATE}}}
\where
\draline{PRE2}{\text{\expression{\term{name?} \in \set{known}}}}\\
\draline{O1}{\text{\expression{\term{date!} =
\term{\set{birthday} (\term{name?})}}}}
\end{schema}}

\allows{PRE2}{O1}
\uses{OS1}{SS1}
\requires{OS1}{PRE2}

\begin{zed}
\set{REPORT} ::= \term{ok} | \term{already\_known} | \term{not\_known}
\end{zed}

\draschema{OS3}{
\begin{schema}{Success}
\text{\declaration{\term{result!}: \expression{REPORT}}}
\where
\draline{O3}{\text{\expression{\term{result!} = \term{ok}}}}
\end{schema}}

\requires{OS3}{O3}
\uses{OS3}{SS1}

\draschema{OS4}{
\begin{schema}{AlreadyKnown}
\text{\Xi BirthdayBook} \\
\text{\declaration{\term{name?}: \expression{NAME}}} \\
\text{\declaration{\term{result!}: \expression{REPORT}}}
\where
\draline{PRE3}{\text{\expression{\term{name?} \in \set{known}}}} \\
\draline{O4}{\text{\expression{\term{result!} = \term{already\_known}}}}
\end{schema}}

\requires{OS4}{PRE3}
\allows{PRE3}{O4}
\uses{OS4}{SS1}

\draschema{OS5}{
\begin{schema}{NotKnown}
\text{\Xi BirthdayBook} \\
\text{\declaration{\term{name?}:\expression{NAME}}} \\
\text{\declaration{\term{result!}:\expression{REPORT}}}
\where
\draline{PRE4}{\text{\expression{\term{name?} \notin \set{known}}}} \\
\draline{O5}{\text{\expression{\term{result!} = \term{not\_known}}}}
\end{schema}}

\requires{OS5}{PRE4}
\allows{PRE4}{O5}
\uses{OS5}{SS1}

\begin{zed}
\draline{TS1}{RAddBirthday == \text{\expression{(\text{AddBirthday}
\land \text{Success})\\  \lor \text{AlreadyKnown}}}} \\
\draline{TS2}{RFindBirthday == \text{\expression{(\text{FindBirthday}
 \land \text{Success}) \lor \text{NotKnown}}}}
\end{zed}

\totalises{TS1}{CS1}
\totalises{TS1}{OS3}
\totalises{TS1}{OS4}
\totalises{TS2}{OS1}
\totalises{TS2}{OS3}
\totalises{TS2}{OS5}

}
\end{document}
```

A.2.8 ZCGa and ZDRa output



A.2.9 Dependency and Goto Graphs





A.2.10 General Proof Skeleton

stateSchema SS1

stateInvarients SI1

initialSchema IS1

postcondition PO2

outputSchema OS1

precondition PRE2

changeSchema CS1

precondition PRE1

outputSchema OS5

precondition PRE4

outputSchema OS4

precondition PRE3

postcondition PO3

output O5

output O4

outputSchema OS3

output O3

output O1

totaliseSchema TS2

totaliseSchema TS1

lemma L1_(CS1)

A.2.11 Isabelle Proof Skeleton

```
theory birthdaybook
imports
Main

begin
(*DATATYPES*)

record SS1 =
(*DECLARATIONS*)

locale 1n2 =
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 CS1 ::
"(*CS1_TYPES*) => bool"
where
"CS1 (*CS1_VARIABLES*) ==
 (PRE1)
∧ (PO3)"
```

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

definition OS4 ::
 "(*OS4_TYPES*) => bool"
where
"OS4 (*OS4_VARIABLES*) == (PRE3)
∧ (O4)"

definition OS3 ::
 "(*OS3_TYPES*) => bool"
where
"OS3 (*OS3_VARIABLES*) == (O3)"

definition TS2 ::
"(*TS2_TYPES*) => bool"
where
"TS2 (*TS2_VARIABLES*) == (*TS2_EXPRESSION*)"

definition TS1 ::
"(*TS1_TYPES*) => bool"
where
"TS1 (*TS1_VARIABLES*) == (*TS1_EXPRESSION*)"

lemma CS1_L1:
"(∃ (*CS1_VARIABLESANDTYPES*).
(PRE1)
∧ (PO3)
∧ (SI1)
∧ (SI1'))"
sorry

end
end
```

A.2.12 Isabelle Filled In

```
theory new5
imports
Main

begin
typedecl NAME
typedecl DATE
datatype REPORT = ok | already_known | not_known


record BirthdayBook =
BIRTHDAY :: "(NAME ⇀ DATE)"
KNOWN :: "(NAME set)"


locale gpsabirthdaybook =
fixes birthday :: "(NAME ⇀ DATE)"
and known :: "(NAME set)"

assumes
"(known = dom birthday)"
begin

definition InitBirthdayBook ::
 "BirthdayBook ⇒ (NAME set) ⇒ (NAME ⇀ DATE) ⇒ BirthdayBook => bool"
where
"InitBirthdayBook birthdaybook' known' birthday' birthdaybook == (known' = {})"

definition FindBirthday ::
 "(NAME set) ⇒ NAME ⇒ BirthdayBook ⇒ BirthdayBook ⇒ DATE ⇒ (NAME ⇀ DATE) => bool"
where
"FindBirthday known' name birthdaybook birthdaybook' date birthday' == (name ∈ known)
∧ (date = the (birthday name))"
```

```
definition AddBirthday ::
"(NAME set) ⇒ NAME ⇒ BirthdayBook ⇒ BirthdayBook ⇒ DATE ⇒ (NAME ⇀ DATE) => bool"
where
"AddBirthday known' name birthdaybook birthdaybook' date birthday' ==
 (name ∉ known)
∧ (birthday' = birthday (name ↦ date ))"

definition NotKnown ::
 "(NAME set) ⇒ NAME ⇒ BirthdayBook ⇒ BirthdayBook ⇒ (NAME ⇀ DATE) ⇒ REPORT => bool"
where
"NotKnown known' name birthdaybook birthdaybook' birthday' result == (name ∉ known)
∧ (result = not_known)"

definition AlreadyKnown ::
 "(NAME set) ⇒ NAME ⇒ BirthdayBook ⇒ BirthdayBook ⇒ (NAME ⇀ DATE) ⇒ REPORT => bool"
where
"AlreadyKnown known' name birthdaybook birthdaybook' birthday' result == (name ∈ known)
∧ (result = already_known)"

definition Success ::
 "REPORT => bool"
where
"Success result == (result = ok)"

definition RFindBirthday ::
"(NAME set) ⇒ NAME ⇒ BirthdayBook ⇒ BirthdayBook ⇒ DATE ⇒ (NAME ⇀ DATE) ⇒ REPORT => bool"
where
"RFindBirthday known' name birthdaybook birthdaybook' date birthday' result ==
(((FindBirthday known' name birthdaybook birthdaybook' date birthday') ∧
 (Success result)) ∨
 (NotKnown known' name birthdaybook birthdaybook' birthday' result))"
```

```
definition RAddBirthday ::
"(NAME set) ⇒ NAME ⇒ BirthdayBook ⇒ BirthdayBook ⇒ DATE ⇒ (NAME ⇸ DATE) ⇒ REPORT => bool"
where
"RAddBirthday known' name birthdaybook birthdaybook' date birthday' result ==
(((AddBirthday known' name birthdaybook birthdaybook' date birthday') ∧
 (Success result)) ∨
 (AlreadyKnown known' name birthdaybook birthdaybook' birthday' result))"

lemma AddBirthday_L1:
"(∃ known' :: (NAME set).
∃ name :: NAME.
∃ date :: DATE.
∃ birthday' :: (NAME ⇸ DATE).
∃ birthday :: (NAME ⇸ DATE).
∃ known :: (NAME set).
(name ∉ known)
∧ (birthday' = birthday (name ↦ date ))
∧ (known = dom birthday)
∧(known' = dom birthday'))"
sorry

end
end
```

A.2.13 Full Proof in Isabelle

```
theory new6
imports
Main

begin
typedecl NAME
typedecl DATE
datatype REPORT = ok | already_known | not_known

record BirthdayBook =
BIRTHDAY :: "(NAME ⇸ DATE)"
KNOWN :: "(NAME set)"

locale gpsabirthdaybook =
fixes birthday :: "(NAME ⇸ DATE)"
and known :: "(NAME set)"

assumes
"(known = dom birthday)"
begin

definition InitBirthdayBook ::
 "BirthdayBook ⇒ (NAME set) ⇒ (NAME ⇸ DATE) ⇒ BirthdayBook => bool"
where
"InitBirthdayBook birthdaybook' known' birthday' birthdaybook == (known' = {})"

definition FindBirthday ::
 "(NAME set) ⇒ NAME ⇒ BirthdayBook ⇒ BirthdayBook ⇒ DATE ⇒ (NAME ⇸ DATE) => bool"
where
"FindBirthday known' name birthdaybook birthdaybook' date birthday' == (name ∈ known)
∧ (date = the (birthday name))"
```

```
definition AddBirthday ::
"(NAME set) ⇒ NAME ⇒ BirthdayBook ⇒ BirthdayBook ⇒ DATE ⇒ (NAME ⇸ DATE) => bool"
where
"AddBirthday known' name birthdaybook birthdaybook' date birthday' ==
 (name ∉ known)
∧ (birthday' = birthday (name ↦ date ))"

definition NotKnown ::
 "(NAME set) ⇒ NAME ⇒ BirthdayBook ⇒ BirthdayBook ⇒ (NAME ⇸ DATE) ⇒ REPORT => bool"
where
"NotKnown known' name birthdaybook birthdaybook' birthday' result == (name ∉ known)
∧ (result = not_known)"

definition AlreadyKnown ::
 "(NAME set) ⇒ NAME ⇒ BirthdayBook ⇒ BirthdayBook ⇒ (NAME ⇸ DATE) ⇒ REPORT => bool"
where
"AlreadyKnown known' name birthdaybook birthdaybook' birthday' result == (name ∈ known)
∧ (result = already_known)"

definition Success ::
 "REPORT => bool"
where
"Success result == (result = ok)"

definition RFindBirthday ::
"(NAME set) ⇒ NAME ⇒ BirthdayBook ⇒ BirthdayBook ⇒ DATE ⇒ (NAME ⇸ DATE) ⇒ REPORT => bool"
where
"RFindBirthday known' name birthdaybook birthdaybook' date birthday' result ==
(((FindBirthday known' name birthdaybook birthdaybook' date birthday') ∧
 (Success result)) ∨
 (NotKnown known' name birthdaybook birthdaybook' birthday' result))"
definition RAddBirthday ::
"(NAME set) ⇒ NAME ⇒ BirthdayBook ⇒ BirthdayBook ⇒ DATE ⇒ (NAME ⇸ DATE) ⇒ REPORT => bool"
where
"RAddBirthday known' name birthdaybook birthdaybook' date birthday' result ==
 (((AddBirthday known' name birthdaybook birthdaybook' date birthday') ∧
 (Success result)) ∨
 (AlreadyKnown known' name birthdaybook birthdaybook' birthday' result))"

lemma AddBirthday_L1:
"(∃ known' :: (NAME set).
∃ name :: NAME.
∃ date :: DATE.
∃ birthday' :: (NAME ⇸ DATE).
∃ birthday :: (NAME ⇸ DATE).
∃ known :: (NAME set).
(name ∉ known)
∧ (birthday' = birthday (name ↦ date ))
∧ (known = dom birthday)
∧(known' = dom birthday'))"
by auto

(*Here I add my own properties about the birthdaybook specification *)

definition (in gpsabirthdaybook)
birthdaybookstate :: "BirthdayBook ⇒ bool"
where
"birthdaybookstate birthdaybook == (known = dom birthday)"
```

```
lemma AddBirthdayIsHonest:
"(∃ known' birthday' birthdaybook birthdaybook' date.
AddBirthday known' name birthdaybook birthdaybook' date birthday')
⟷
(name ∉ known)"
apply (unfold AddBirthday_def)
apply auto
done

lemma preAddBirthdayTotal:
" (name ∉ known) ∨ (name ∈ known)"
apply (rule excluded_middle)
done

lemma BirthdayBookPredicate:
"(∃ birthdaybook. birthdaybookstate birthdaybook)
⟶ known = dom birthday"
apply (rule impI)
apply (unfold birthdaybookstate_def)
apply auto
done

lemma InitisOk:
"(∃ birthdaybook. InitBirthdayBook birthdaybook' known' birthday' birthdaybook)
⟷ (known' = {}) "
apply (unfold InitBirthdayBook_def)
apply auto
done

lemma RAddBirthdayIsTotal:
"(∃ known' birthday' birthdaybook
birthdaybook' date.
RAddBirthday known' name birthdaybook birthdaybook' date birthday' result)
⟶
(name ∉ known) ∨ (name ∈ known)"
apply (unfold RAddBirthday_def)
apply (unfold AddBirthday_def AlreadyKnown_def Success_def)
apply auto
done

end
end
```

A.3 An example of a specification which fails ZCGa but passes ZDRa

This section shows an example of a specification which is rhetorically correct and passes the ZDRa check however the grammar of the specification is incorrect and therefore fails the ZCGa check. We input the compiled output for each of the examples. For reference to the code the reader is directed to [15].

A.3.1 Raw Latex

```
\documentclass{article}
\usepackage{zmathlang}

\begin{document}

\begin{zed}
[NAME].
\end{zed}

\begin{zed}
[SURNAME]
\end{zed}

\begin{schema}{TelephoneDirectory}
persons: NAME \fun SURNAME \\.
phoneNumbers: NAME \pfun TELEPHONE
\where.
\dom phoneNumbers=\dom persons
\end{schema}

\begin{schema}{InitTelephoneDirectory}.
TelephoneDirectory'.
\where.
phoneNumbers' = \{\} \\
persons' = \{\}
\end{schema}

\begin{schema}{AddPerson}
TheTelephoneDirectory \\
name?: NAME \\
surname?: SURNAME \\
\where
name? \mapsto surname? \notin persons\\
persons' = persons \cup \{name? \mapsto surname? \}
\end{schema}

\begin{schema}{AddNumber}
\Delta TelephoneDirectory \\
n?: NAME \\
s?: SURNAME \\
p?: TELEPHONE
\where
n \mapsto s \in persons\\
p? \notin \ran phoneNumbers \\
phoneNumbers' = phoneNumbers \cup \{n \mapsto phone?\}
\end{schema}

\begin{schema}{RemovePerson}
\Delta TelephoneDirectory \\
n?: NAME \\
s?: SURNAME \\
p?: TELEPHONE
\where
n? \mapsto s? \in persons\\
n? \mapsto p? \notin phoneNumbers \\
persons' = persons \setminus \{n? \mapsto s?\}
\end{schema}

\begin{zed}
OUTPUT ::= success | alreadyInDirectory | nameNotInDirectory | numberInUse
\end{zed}

\begin{schema}{Success}
message!:OUTPUT
\where
message! = success
\end{schema}
```

```
\begin{schema}{AlreadyInDirectory}
message!:OUTPUT \\
n?: NAME \\
s?: SURNAME
\where
n? \mapsto s \in person \\
message! = alreadyInDirectory
\end{schema}

\begin{schema}{NameNotInDirectory}
message!:OUTPUT \\
n?: NAME \\
s?: SURNAME
\where
n? \mapsto s? \notin persons \\
message! = NotInDirectory
\end{schema}

\begin{schema}{NumberInUse}
message!:OUTPUT \\
p?: TELEPHONE
\where
p? \in \ran phoneNumbers \\
message! = numberInUse
\end{schema}

\begin{zed}
TotalAddPerson \defs (AddPerson \land Success).
\lor AlreadyInDirectory \\
TotalRemovePerson \defs (RemovePerson \land Success)
. \lor NameNotInDirectory \\
TotalAddNumber \defs (AddNumber \land Success).
\lor NameNotInDirectory \lor NumberInUse \\
\end{zed}

\end{document}
```

A.3.2 Raw Latex output

$[NAME]$

$[SURNAME]$

TelephoneDirectory

$$persons : NAME \rightarrow SURNAME$$
$$phoneNumbers : NAME \nrightarrow TELEPHONE$$
$$\operatorname{dom} phoneNumbers = \operatorname{dom} persons$$

InitTelephoneDirectory

$$TelephoneDirectory'$$
$$phoneNumbers' = \{\}$$
$$persons' = \{\}$$

AddPerson

$$TheTelephoneDirectory$$
$$name? : NAME$$
$$surname? : SURNAME$$
$$name? \mapsto surname? \notin persons$$
$$persons' = persons \cup \{name? \mapsto surname?\}$$

AddNumber

$$\Delta TelephoneDirectory$$
$$n? : NAME$$
$$s? : SURNAME$$
$$p? : TELEPHONE$$
$$n \mapsto s \in persons$$
$$p? \notin \operatorname{ran} phoneNumbers$$
$$phoneNumbers' = phoneNumbers \cup \{n \mapsto phone?\}$$

$$OUTPUT ::= success \mid alreadyInDirectory \mid nameNotInDirectory \mid numberInUse$$

Success

$$message! : OUTPUT$$

$$message! = success$$

AlreadyInDirectory

$$message! : OUTPUT$$
$$n? : NAME$$
$$s? : SURNAME$$

$$n? \mapsto s \in person$$
$$message! = alreadyInDirectory$$

NameNotInDirectory

$$message! : OUTPUT$$
$$n? : NAME$$
$$s? : SURNAME$$

$$n? \mapsto s? \notin persons$$
$$message! = NotInDirectory$$

NumberInUse

$$message! : OUTPUT$$
$$p? : TELEPHONE$$

$$p? \in \operatorname{ran} phoneNumbers$$
$$message! = numberInUse$$

$$TotalAddPerson \mathrel{\widehat=} (AddPerson \land Success) \lor AlreadyInDirectory$$
$$TotalRemovePerson \mathrel{\widehat=} (RemovePerson \land Success) \lor NameNotInDirectory$$
$$TotalAddNumber \mathrel{\widehat=} (AddNumber \land Success) \lor NameNotInDirectory \lor NumberInUse$$

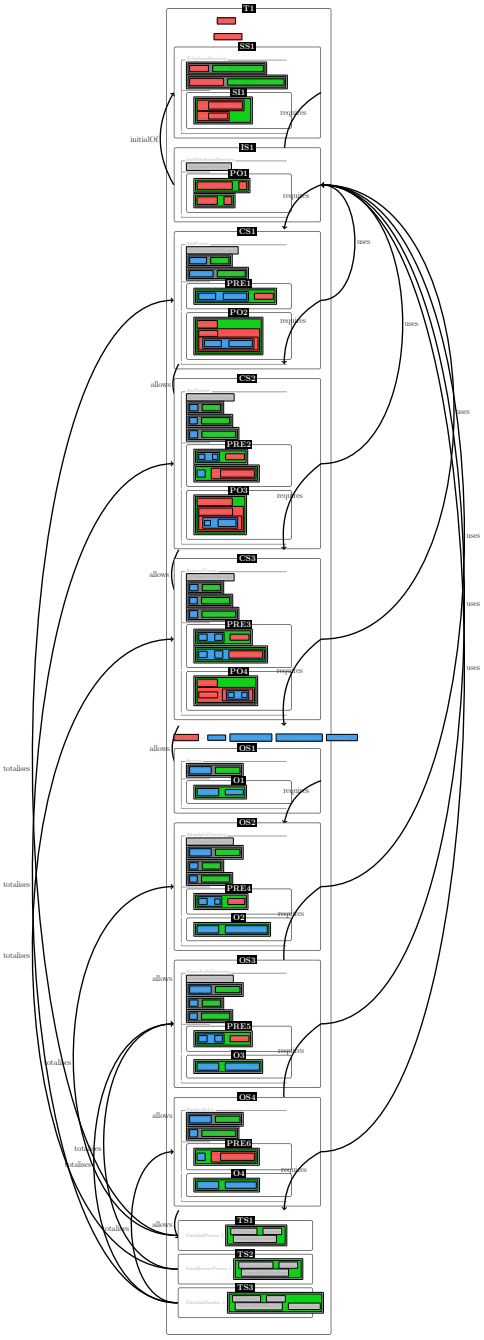
A.3.3 ZCGa and ZDRa Annotated Latex Code

```
\documentclass{article}
\usepackage{zmathlang}

\begin{document}

\dratheory{T1}{0.2}{

\begin{zed}
[\set{NAME}].
\end{zed}

\begin{zed}
[\set{SURNAME}]
\end{zed}

\draschema{SS1}{
\begin{schema}{TelephoneDirectory}
\text{\declaration{\set{persons}: \expression{NAME \fun SURNAME}}} \\.
\text{\declaration{\set{phoneNumbers}: \expression{NAME \pfun TELEPHONE}}}
\where.
\draline{SI1}{
\text{\expression{\set{\dom \set{phoneNumbers}} = \set {\dom \set{persons}}}}}
\end{schema}}

\requires{SS1}{SI1}

\draschema{IS1}{
\begin{schema}{InitTelephoneDirectory}.
\text{TelephoneDirectory' }
\where.
\draline{PO1}{
\text{\expression{\set{phoneNumbers'} = \set{\{\}}}} \\
\text{\expression{\set{persons'} = \set{\{\}}}}}
\end{schema}}

\initialof{IS1}{SS1}
\requires{IS1}{PO1}

\draschema{CS1}{
\begin{schema}{AddPerson}
\text{TheTelephoneDirectory} \\
\text{\declaration{\term{name?}: \expression{NAME}}} \\
\text{\declaration{\term{surname?}: \expression{SURNAME}}} \\
\where
\draline{PRE1}{
\text{\expression{\term{\term{name?} \mapsto \term{surname?}} \notin \set{persons}}}}\\
\draline{PO2}{
\text{\expression{\set{persons'} = \set{\set{persons} \cup.
\set{\{\term{\term{name?} \mapsto \term{surname?}}\}}}}}}
\end{schema}}

\uses{CS1}{IS1}
\requires{CS1}{PRE1}
\allows{PRE1}{PO2}

\draschema{CS2}{
\begin{schema}{AddNumber}
\text{\Delta TelephoneDirectory} \\
\text{\declaration{\term{n?}: \expression{NAME}}} \\
\text{\declaration{\term{s?}: \expression{SURNAME}}} \\
\text{\declaration{\term{p?}: \expression{TELEPHONE}}}
\where
\draline{PRE2}{
\text{\expression{\term{\term{n} \mapsto \term{s}} \in \set{persons}}}\\
\text{\expression{\term{p?} \notin \set{\ran \set{phoneNumbers}}}}} \\
\draline{PO3}{
\text{\expression{\set{phoneNumbers'} = \set{\set{phoneNumbers}.
\cup \set{\{\term{\term{n} \mapsto \term{phone?}}\}}}}}
\end{schema}}

\uses{CS2}{IS1}
\allows{PRE2}{PO3}
\requires{CS2}{PRE2}
```

```
\draschema{CS3}{
\begin{schema}{RemovePerson}
\text{\Delta TelephoneDirectory} \\
\text{\declaration{\term{n?}: \expression{NAME}}} \\
\text{\declaration{\term{s?}: \expression{SURNAME}}} \\
\text{\declaration{\term{p?}: \expression{TELEPHONE}}}
\where
\draline{PRE3}{
\text{\expression{\term{\term{n?} \mapsto \term{s?}} \in \set{persons}}} \\
\text{\expression{\term{\term{n?} \mapsto \term{p?} \notin \set{phoneNumbers}}}}}\\
\draline{PO4}{
\text{\expression{\set{persons'} = \set{\set{persons} \setminus
\set{\{\term{\term{n} \mapsto \term{s}}\}}}}}}
\end{schema}}

\uses{CS3}{IS1}
\allows{PRE3}{PO4}
\requires{CS3}{PRE3}

\begin{zed}
\set{OUTPUT} ::= \term{success} | \term{alreadyInDirectory} |
\term{nameNotInDirectory} | \term{numberInUse}
\end{zed}

\draschema{OS1}{
\begin{schema}{Success}
\text{\declaration{\term{message!}: \expression{OUTPUT}}} \\
\where
\draline{O1}{
\text{\expression{\term{message!} = \term{success}}}}
\end{schema}}

\requires{OS1}{O1}

\draschema{OS2}{
\begin{schema}{AlreadyInDirectory}
\text{\Xi TelephoneDirectory} \\
\text{\declaration{\term{message!}: \expression{OUTPUT}}} \\
\text{\declaration{\term{n?}: \expression{NAME}}} \\
\text{\declaration{\term{s?}: \expression{SURNAME}}}
\where
\draline{PRE4}{
\text{\expression{\term{\term{n?} \mapsto \term{s}} \in \set{person}}}} \\
\draline{O2}{
\text{\expression{\term{message!} = \term{alreadyInDirectory}}}}
\end{schema}}

\draschema{OS3}{
\begin{schema}{NameNotInDirectory}
\text{\Xi TelephoneDirectory} \\
\text{\declaration{\term{message!}: \expression{OUTPUT}}} \\
\text{\declaration{\term{n?}: \expression{NAME}}} \\
\text{\declaration{\term{s?}: \expression{SURNAME}}}
\where
\draline{PRE5}{
\text{\expression{\term{\term{n?} \mapsto \term{s?}} \notin \set{persons}}}} \\
\draline{O3}{
\text{\expression{\term{message!} = \term{notInDirectory}}}}
\end{schema}}

\requires{OS3}{PRE5}
\allows{PRE5}{O3}
\uses{OS3}{IS1}

\draschema{OS4}{
\begin{schema}{NumberInUse}
\text{\declaration{\term{message!}: \expression{OUTPUT}}} \\
\text{\declaration{\term{p?}: \expression{TELEPHONE}}}
\where
\draline{PRE6}{
\text{\expression{\term{p?} \in \set{ \ran \set{phoneNumbers}}}}} \\
\draline{O4}{\text{\expression{\term{message!} = \term{numberInUse}}}}
\end{schema}}
```

```
\requires{OS4}{PRE6}
\allows{PRE6}{O4}
\uses{OS4}{IS1}

\begin{zed}
\draline{TS1}{TotalAddPerson \defs \text{\expression{
(\text{AddPerson} \land \text{Success}) \\ \lor \text{AlreadyInDirectory}}}} \\
\draline{TS2}{TotalRemovePerson \defs \text{\expression{
(\text{RemovePerson} \land \text{Success}) \\ \lor \text{NameNotInDirectory}}}} \\
\draline{TS3}{TotalAddNumber \defs \text{\expression{
(\text{AddNumber} \land \text{Success}) \\ \lor \text{NameNotInDirectory}
. \lor \text{NumberInUse}}}} \\
\end{zed}

\totalises{TS1}{CS1}
\totalises{TS1}{OS2}
\totalises{TS2}{CS3}
\totalises{TS2}{OS3}
\totalises{TS3}{CS2}
\totalises{TS3}{OS4}
\totalises{TS3}{OS3}

}

\end{document}
```

A.3.4 ZCGa and ZDRa output



A.3.5 Messages when running the specification through the ZCGa and ZDRa checks

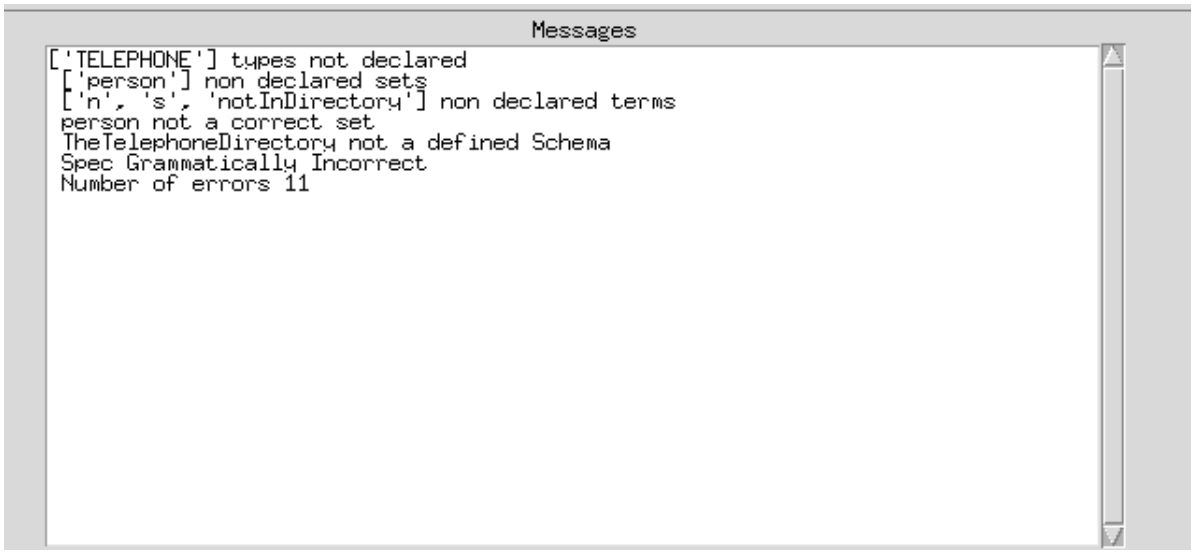


Figure A.1: Message when checking the specification for ZCGa correctness.

Figure A.2: Message when checking the specification for ZDRa correctness.

A.4 An example of a specification which fails ZDRa but passes ZCGa

This section shows an example of a specification which is grammatically correct and passes the ZCGa check however there are loops in it's rhetorical reasoning

and therefore fails the ZDRa check. We input the compiled output for each of the examples. For reference to the code the reader is directed to [15].

A.4.1 Raw Latex output

$[NAME]$

$[SURNAME]$

$[TELEPHONE]$

TelephoneDirectory

$$phoneNumbers : NAME \rightarrow SURNAME$$
$$phoneNumbers : NAME \nrightarrow TELEPHONE$$

$$\mathrm{dom}\, phoneNumbers = \mathrm{dom}\, persons$$

InitTelephoneDirectory

$$TelephoneDirectory'$$

$$phoneNumbers' = \{\}$$
$$persons' = \{\}$$

AddPerson

$$\Delta TelephoneDirectory$$
$$name? : NAME$$
$$surname? : SURNAME$$
$$phone? : TELEPHONE$$

$$name? \mapsto surname? \notin persons$$
$$persons' = persons \cup \{name? \mapsto surname?\}$$

AddNumber

$$\Delta TelephoneDirectory$$
$$n? : NAME$$
$$s? : SURNAME$$
$$p? : TELEPHONE$$

$$n? \mapsto s? \in persons$$
$$p? \notin \mathrm{ran}\, phoneNumbers$$
$$phoneNumbers' = phoneNumbers \cup \{name? \mapsto phone?\}$$

RemovePerson

$$\Delta TelephoneDirectory$$
$$n? : NAME$$
$$s? : SURNAME$$
$$p? : TELEPHONE$$

$$n? \mapsto s? \in persons$$
$$n? \mapsto p? \notin phoneNumbers$$
$$persons' = persons \setminus \{n? \mapsto s?\}$$

RemoveNumber

$\Delta TelephoneDirectory$
$p? : TELEPHONE$

$p? \in \operatorname{ran} phoneNumbers$
$\exists m : \operatorname{dom} persons \bullet$
$m \mapsto p? \in phoneNumbers \land$
$phoneNumbers' = phoneNumbers \setminus \{m \mapsto p?\}$

$OUTPUT ::= success \mid alreadyInDirectory \mid nameNotInDirectory \mid numberInUse \mid numberDoesntExist$

Success

$message! : OUTPUT$

$message! = success$

AlreadyInDirectory

$\Xi TelephoneDirectory$
$message! : OUTPUT$
$n? : NAME$
$s? : SURNAME$

$n? \mapsto s? \in persons$
$message! = alreadyInDirectory$

NameNotInDirectory

$\Xi TelephoneDirectory$
$message! : OUTPUT$
$n? : NAME$
$s? : SURNAME$

$n? \mapsto s? \notin persons$
$message! = nameNotInDirectory$

NumberInUse

$message! : OUTPUT$
$p? : TELEPHONE$

$p? \in \operatorname{ran} phoneNumbers$
$message! = numberInUse$

NumberDoesntExist

$message! : OUTPUT$
$p? : TELEPHONE$

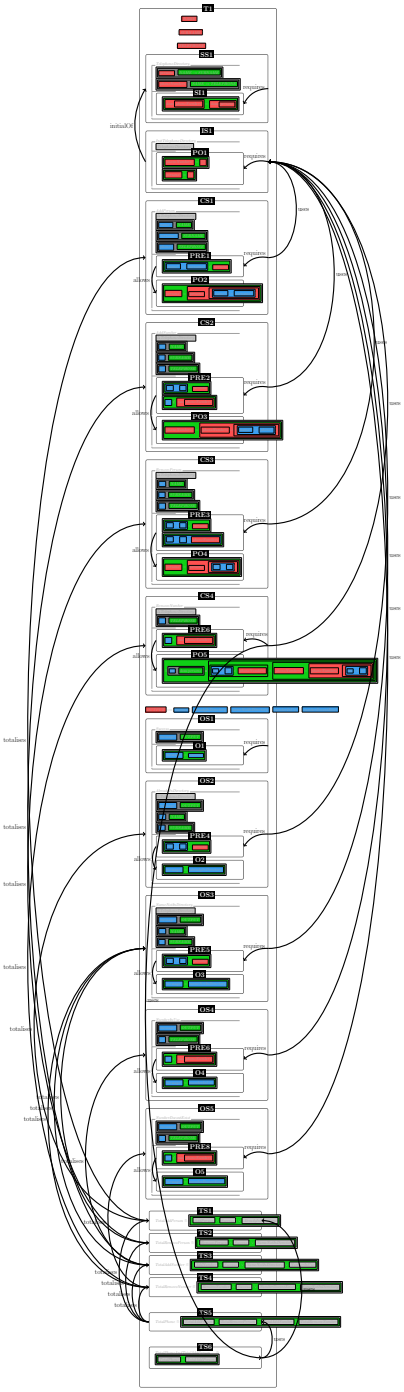
$p? \notin \operatorname{ran} phoneNumbers$
$message! = numberDoesntExist$

$TotalAddPerson \mathrel{\widehat=} (AddPerson \land Success) \lor AlreadyInDirectory$
$TotalRemovePerson \mathrel{\widehat=} (RemovePerson \land Success)$
$\lor NameNotInDirectory$
$TotalAddNumber \mathrel{\widehat=} (AddNumber \land Success)$
$\lor NameNotInDirectory \lor NumberInUse$
$TotalRemoveNumber \mathrel{\widehat=} (RemoveNumber \land Success)$
$\lor NumberDoesntExist \lor NameNotInDirectory$

$TotalPhone \mathrel{\widehat=} TotalAddPerson \lor TotalRemovePerson$
$\lor TotalAddNumber \lor TotalRemoveNumber$

$TotalPhoneAndTotalAddPerson \mathrel{\widehat=} TotalPhone \lor TotalAddPerson$

A.4.2 ZCGa and ZDRa output



A.4.3 Messages when running the specification through the ZCGa and ZDRa checks

Figure A.3: Message when checking the specification for ZCGa correctness.

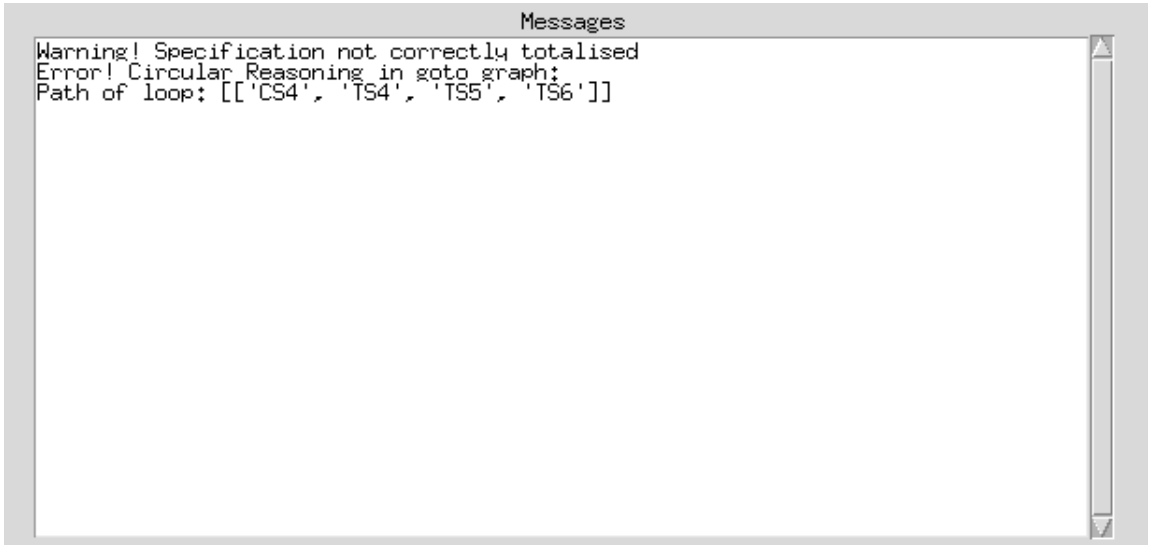


Figure A.4: Message when checking the specification for ZDRa correctness.

A.5 An example of a specification which is semi formal

This section shows an auto pilot specification which is partially written in natural language and partially written formally. Thus it is a natural langauge specification which is on it's way to being formalised.

A.5.1 Raw Latex output

1. The mode-control panel contains four buttons for selecting modes and three displays for dialing in or displaying values. The system supports the following four modes:

   - attitude control wheel steering (att_cws)
   - flight path angle selected (fpa_sel)
   - altitude engage (alt_eng)
   - calibrated air speed (cas_eng)

   $events ::= press\_att\_cws \mid press\_cas\_eng \mid press\_alt\_eng \mid press\_fpa\_sel$

   Only one of the first three modes can be engaged at any time. However, the cas_eng mode can be engaged at the same time as any of the other modes. The pilot engages a mode by pressing the corresponding button on the panel. One of the three modes, att_cws, fpa_sel, or alz_eng, should be engaged at all times. Engaging any of the first three modes will automatically cause the other two to be disengaged since only one of these three modes can be engaged at a time.

   $mode\_status ::= off \mid engaged$

   off_eng

   $$mode : mode\_status$$

   $$mode = off \lor mode = engaged$$

   AutoPilot

   $$att\_cws : mode\_status$$
   $$fpa\_sel : mode\_status$$
   $$alt\_eng : mode\_status$$
   $$cas\_eng : mode\_status$$

   att_cwsDo

   $$\Delta AutoPilot$$

   $$att\_cws = off$$
   $$att\_cws' = engaged$$
   $$fpa\_sel' = off$$
   $$alt\_eng' = off$$
   $$cas\_eng' = off \lor engaged$$

2. There are three displays on the panel: and altitude [ALT], flight path angle [FPA], and calibrated air speed [CAS]. The displays usually show the current values for the altitude, flight path angle, and air speed of the aircraft. However, the pilot can enter a new value into a display by dialing in the value using the knob next to the display. This is the target or "pre-selected" value that the pilot wishes the aircraft to attain. For example, if the pilot wishes to climb to 25,000 feet, he will dial 25,000 into the altitude display window and then press the alz_eng button to engage the altitude mode. Once the target value is achieved or the mode is disengaged, the display reverts to showing the "current" value.

3. If the pilot dials in an altitude that is more than 1,200 feet above the current altitude and then presses the alz_eng button, the altitude mode will not directly engage. Instead, the altitude engage mode will change to "armed" and the flight-path angle select mode is engaged. The pilot must then dial in a flight-path angle for the flight-control system to follow until the aircraft attains the desired altitude. The flight-path angle select mode will remain engaged until the aircraft is within 1,200 feet of the desired altitude, then the altitude engage mode is automatically engaged.

4. The calibrated air speed and the flight-path angle values need not be pre-selected before the corresponding modes are engaged–the current values displayed will be used. The pilot can dial-in a different target value after the mode is engaged. However, the altitude must be pre-selected before the altitude engage button is pressed. Otherwise, the command is ignored.

5. The calibrated air speed and flight-path angle buttons toggle on and off every time they are pressed. For example, if the calibrated air speed button is pressed while the system is already in calibrated air speed mode that mode will be disengaged. However, if the attitude control wheel steering button is pressed while the attitude control wheel steering mode is already engaged, the button is ignored. Likewise, pressing the altitude engage button while the system is already in altitude engage mode has no effect.

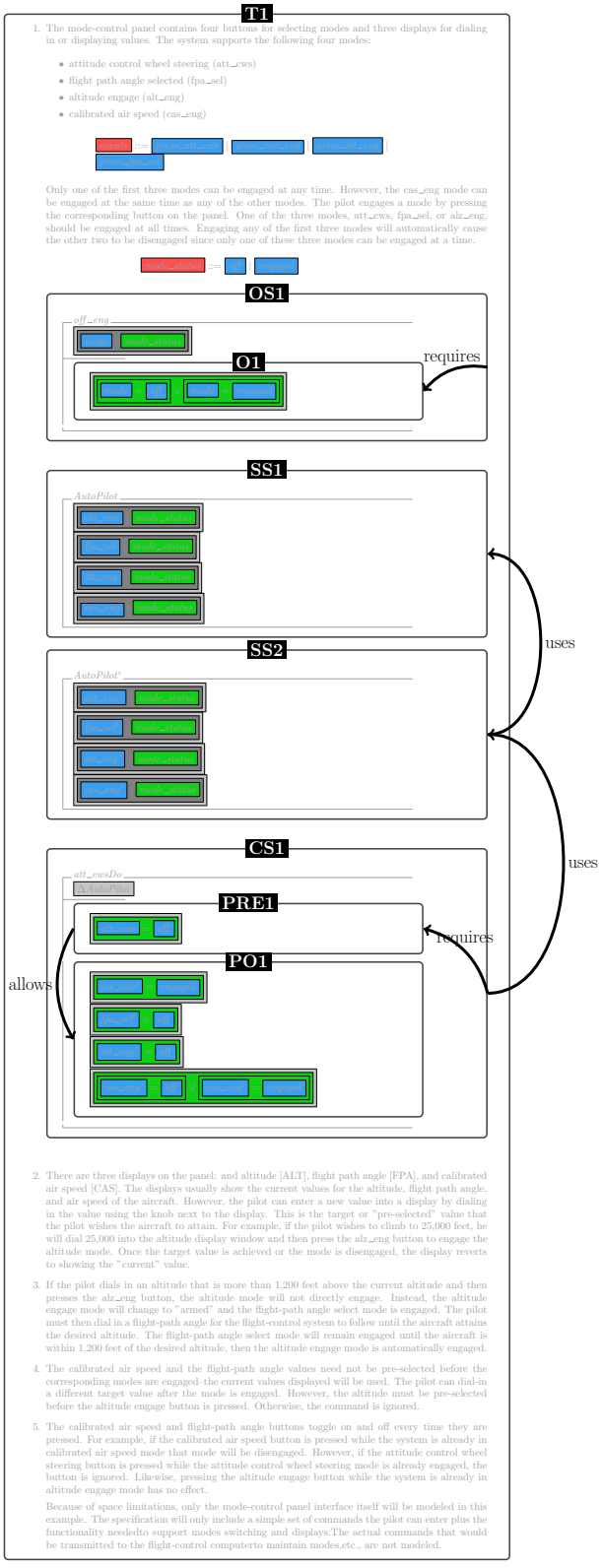
   Because of space limitations, only the mode-control panel interface itself will be modeled in this example. The specification will only include a simple set of commands the pilot can enter plus the functionality neededto support modes switching and displays.The actual commands that would be transmitted to the flight-control computerto maintain modes,etc., are not modeled.

A.5.2 ZCGa and ZDRa Annotated Latex Code

```
\documentclass{article}
\usepackage{zmathlang}

\begin{document}

\dratheory{T1}{0.4}{

\begin{enumerate}
\item The mode-control panel contains four buttons for selecting modes and three displays for
dialling in or displaying values. The system supports the following four modes:

\begin{itemize}
\item attitude control wheel steering (att\_cws)
\item flight path angle selected (fpa\_sel)
\item altitude engage (alt\_eng)
\item calibrated air speed (cas\_eng)
\end{itemize}

\begin{zed}
\set{events} ::= \term{press\_att\_cws} |  \term{press\_cas\_eng} | \term{press\_alt\_eng} | \\
 \term{press\_fpa\_sel}
\end{zed}

Only one of the first three modes can be engaged at any time. However, the cas\_eng mode
can be engaged at the same time as any of the other modes. The pilot engages a mode by pressing
the corresponding button on the panel. One of the three modes, att\_cws, fpa\_sel, or alz\_eng,
should be engaged at all times. Engaging any of the first three modes will automatically cause the
other two to be disengaged since only one of these three modes can be engaged at a time.

\begin{zed}
\set{mode\_status} ::= \term{off} | \term{engaged}
\end{zed}

\draschema{OS1}{
\begin{schema}{off\_eng}
\text{\declaration{\term{mode}: \expression{mode\_status}}}
\where
\draschema{O1}{
\text{\expression{\expression{\term{mode} = \term{off}} \lor \expression{\term{mode} = \term{engaged}}}}}
\end{schema}}

\requires{OS1}{O1}

\draschema{SS1}{
\begin{schema}{AutoPilot}
\text{\declaration{\term{att\_cws}: \expression{mode\_status}}} \\
\text{\declaration{\term{fpa\_sel}: \expression{mode\_status}}} \\
\text{\declaration{\term{alt\_eng}: \expression{mode\_status}}} \\
\text{\declaration{\term{cas\_eng}: \expression{mode\_status}}}
\end{schema}}
```

```
\draschema{SS2}{
\begin{schema}{AutoPilot'}
\text{\declaration{\term{att\_cws'}: \expression{mode\_status}}} \\
\text{\declaration{\term{fpa\_sel'}: \expression{mode\_status}}} \\
\text{\declaration{\term{alt\_eng'}: \expression{mode\_status}}} \\
\text{\declaration{\term{cas\_eng'}: \expression{mode\_status}}}
\end{schema}}

\uses{SS2}{SS1}

\draschema{CS1}{
\begin{schema}{att\_cwsDo}
\text{\Delta AutoPilot }
\where
\draline{PRE1}{
\text{\expression{\term{att\_cws} = \term{off}}}} \\
\draline{PO1}{
\text{\expression{\term{att\_cws'} = \term{engaged}}} \\
\text{\expression{\term{fpa\_sel'} = \term{off}}} \\
\text{\expression{\term{alt\_eng'} = \term{off}}} \\
\text{\expression{\expression{\term{cas\_eng'} = \term{off}} \lor
 \expression{\term{cas\_eng'} = \term{engaged}}}}}
\end{schema}}

\uses{CS1}{SS2}
\allows{PRE1}{PO1}
\requires{CS1}{PRE1}

\item There are three displays on the panel: and altitude [ALT], flight path angle [FPA], and
calibrated air speed [CAS]. The displays usually show the current values for the altitude, flight
path angle, and air speed of the aircraft. However, the pilot can enter a new value into a display by
dialling in the value using the knob next to the display. This is the target or "pre-selected" value that
the pilot wishes the aircraft to attain. For example, if the pilot wishes to climb to 25,000 feet, he
will dial 25,000 into the altitude display window and then press the alz\_eng button to engage the
altitude mode. Once the target value is achieved or the mode is disengaged, the display reverts to
showing the "current" value.

\item If the pilot dials in an altitude that is more than 1,200 feet above the current altitude and
then presses the alz\_eng button, the altitude mode will not directly engage. Instead, the altitude
engage mode will change to "armed" and the flight-path angle select mode is engaged. The pilot
must then dial in a flight-path angle for the flight-control system to follow until the aircraft attains
the desired altitude. The flight-path angle select mode will remain engaged until the aircraft is within
 1,200 feet of the desired altitude, then the altitude engage mode is automatically engaged.

\item The calibrated air speed and the flight-path angle values need not be pre-selected before the
corresponding modes are engaged--the current values displayed will be used. The pilot can dial-in
a different target value after the mode is engaged. However, the altitude must be pre-selected
before the altitude engage button is pressed. Otherwise, the command is ignored.

Because of space limitations, only the mode-control panel interface itself will be modelled in this
example. The specification will only include a simple set of commands the pilot can enter plus the
functionality needed to support modes switching and displays.The actual commands that would
be transmitted to the flight-control computer to maintain modes,etc., are not modelled.

\end{enumerate}
}
\end{document}
```

A.5.3 ZCGa and ZDRa output



A.5.4 Messages when running the specification through the ZCGa and ZDRa checks

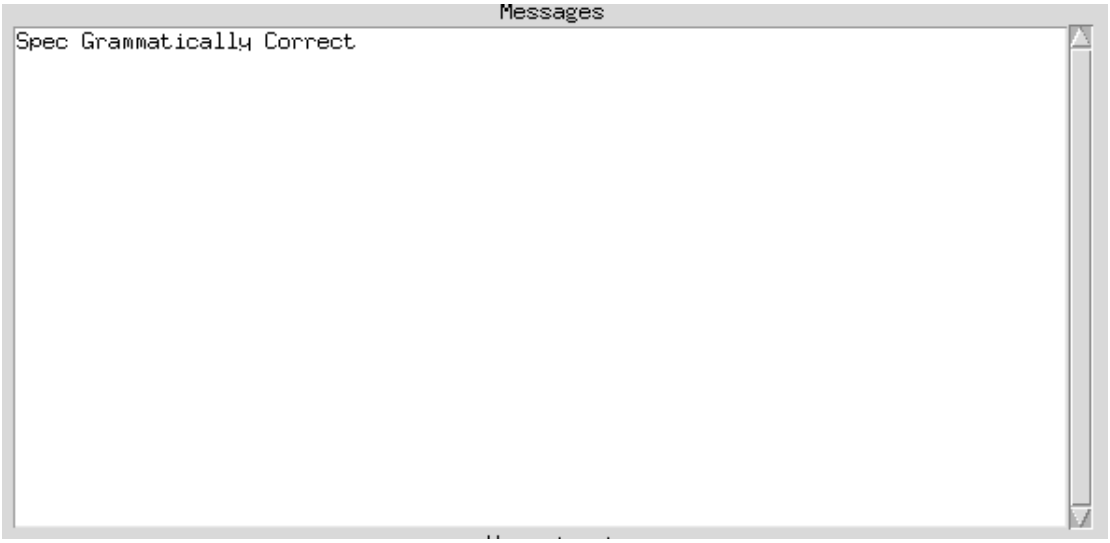


Figure A.5: Message when checking the specification for ZCGa correctness.

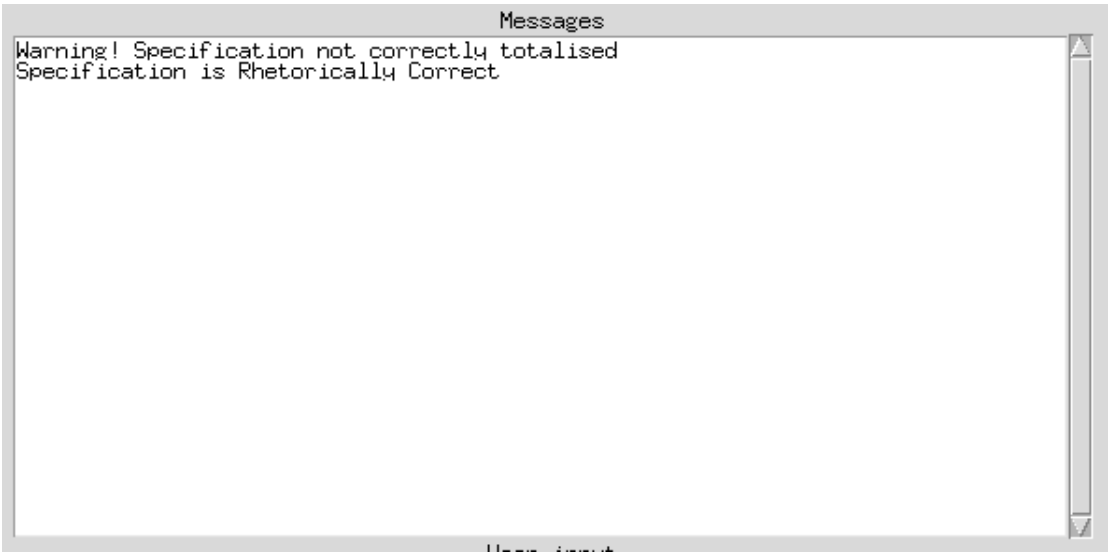


Figure A.6: Message when checking the specification for ZDRa correctness.

A.5.5 General Proof Skeleton

stateSchema SS1

outputSchema OS1

output O1

stateSchema SS2

changeSchema CS1

precondition PRE1

postcondition PO1

A.5.6 Isabelle Proof Skeleton

```
theory gpsa1n2
imports
Main

begin
(*DATATYPES*)

record SS1 =
(*DECLARATIONS*)

locale 1n2 =
fixes (*GLOBAL DECLARATIONS*)
begin

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

definition CS1 ::
"(*CS1_TYPES*) => bool"
where
"CS1 (*CS1_VARIABLES*) ==
 (PRE1)
∧ (PO1)"

end
end
```

A.5.7 Isabelle Filled In

```
theory 5
imports
Main

begin
datatype events = press_att_cws | press_cas_eng | press_alt_eng |
 press_fpa_sel
datatype mode_status = off | engaged


record AutoPilot =
ALT_ENG :: "mode_status"
CAS_ENG :: "mode_status"
ATT_CWS :: "mode_status"
FPA_SEL :: "mode_status"


locale theautopilot =
fixes alt_eng :: "mode_status"
and cas_eng :: "mode_status"
and att_cws :: "mode_status"
and fpa_sel :: "mode_status"

begin

definition off_eng ::
 "mode_status => bool"
where
"off_eng mode == (mode = off ∨ mode = engaged)"

definition att_cwsDo ::
"mode_status ⇒ mode_status ⇒ mode_status ⇒ mode_status => bool"
where
"att_cwsDo fpa_sel' cas_eng' att_cws' alt_eng'  ==
 (att_cws = off)
∧ (att_cws' = engaged)
∧ (fpa_sel' = off)
∧ (alt_eng' = off)
∧ (cas_eng' = off ∨
 cas_eng' = engaged)"

end
end
```

A.6 ModuleReg

A.6.1 ModuleReg Full Proof

This section shows the full proof for the modulereg example from [22]. It includes the filled in Isabelle skeleton. Along with added proofs which have been input by the user.

```
theory new6
imports
Main

begin
typedecl PERSON
typedecl MODULE

record ModuleReg =
STUDENTS :: " PERSON set"
DEGMODULES :: " MODULE set"
TAKING :: "(PERSON * MODULE) set"

locale Themodulereg =
fixes students :: " PERSON set"
and degModules :: " MODULE set"
and taking :: "(PERSON * MODULE) set"
assumes "Domain taking ⊆ students"
 and "Range taking ⊆ degModules"
begin

definition RegForModule ::
"ModuleReg ⇒ ModuleReg ⇒ PERSON ⇒ MODULE => MODULE set ⇒
 PERSON set ⇒ (PERSON * MODULE) set ⇒ bool"
where
"RegForModule modulereg modulereg' p m degModules' students' taking' ==
(p ∈ students)
∧ (m ∈ degModules)
∧ ((p, m) ∉ taking)
∧ (taking' = taking ∪ {(p, m)})
∧ (students' = students)
∧ (degModules' = degModules)"
```

```
definition AddStudent ::
"ModuleReg ⇒ ModuleReg => PERSON ⇒  MODULE set ⇒ PERSON set ⇒
(PERSON * MODULE) set ⇒ bool"
where
"AddStudent modulereg modulereg' p degModules' students' taking' ==
 (
(p ∉ students)
∧ (students' = students ∪ {(p)})
∧ (degModules' = degModules)
∧ (taking' = taking))"

(*The proof obligations generated by ZMathLang start here*)
```

```
lemma RegForModule_L1:
"(∃ degModules:: MODULE set.
∃ students :: PERSON set.
∃ taking :: (PERSON * MODULE) set.
∃ p :: PERSON.
∃ degModules':: MODULE set.
∃ students' :: PERSON set.
∃ taking' :: (PERSON * MODULE) set.
∃ m :: MODULE.
(
(p ∈ students)
∧ (m ∈ degModules)
∧ ((p, m) ∉ taking)
∧ (taking' = taking ∪ {(p, m)})
∧ (students' = students)
∧ (degModules' = degModules))
∧ (Domain taking ⊆ students)
∧ (Range taking ⊆ degModules)
∧ (Domain taking' ⊆ students')
∧ (Range taking' ⊆ degModules')
)"
by (smt Domain_empty Domain_insert Range.intros Range_empty
Range_insert Un_empty Un_insert_right empty_iff empty_subsetI
empty_subsetI insert_mono insert_mono singletonI singletonI
singleton_insert_inj_eq' singleton_insert_inj_eq')

lemma AddStudent_L2:
"(∃ degModules:: MODULE set.
∃ students :: PERSON set.
∃ taking :: (PERSON * MODULE) set.
∃ p :: PERSON.
∃ degModules':: MODULE set.
∃ students' :: PERSON set.
∃ taking' :: (PERSON * MODULE) set.
 (
 (students' = students ∪ {(p)})
∧ (degModules' = degModules)
∧ (taking' = taking))
∧ (Domain taking ⊆ students)
∧ (Range taking ⊆ degModules)
∧ (Domain taking' ⊆ students')
∧ (Range taking' ⊆ degModules')
)"
by blast

(*Here I add other safety properties about the ModuleReg specification
which I wish to prove*)

lemma pre_AddStudent:
"(∃ modulereg modulereg' students' degModules' taking'.
AddStudent modulereg modulereg' p degModules' students' taking')
⟷ (p ∉ students)"
apply (unfold AddStudent_def)
apply auto
done
```

```
lemma pre_RegForModule:
"(∃ modulereg modulereg' students' degModules' taking'.
RegForModule modulereg modulereg' p m degModules' students' taking')
 ⟷ (p ∈ students)
 ∧ (m ∈ degModules)
 ∧ ((p,m) ∉ taking)"
 apply (unfold RegForModule_def)
 apply auto
done

definition InitModuleReg::
"ModuleReg ⇒ PERSON set ⇒ MODULE set ⇒ (PERSON * MODULE) set ⇒ bool"
where
"InitModuleReg modulereg' students' degmodules' taking' == ((
(students' = {})
∧ (degmodules' = {})
∧ (taking' = {})))"

lemma InitOK:
"(∃ modulereg'. InitModuleReg modulereg' students' degmodules' taking')
⟶  ((
(students' = {})
∧ (degmodules' = {}))
∧ ((Domain taking' ⊆ students')
∧ (Range taking' ⊆ degModules'))"
by (simp add: InitModuleReg_def)

lemma RegForModuleNotEmpty:
"(∃ modulereg modulereg' students' degModules' taking' p m.
RegForModule modulereg modulereg' students' degModules' taking' p m)
 ⟶ (students ≠ {})
 ∧ (degModules ≠ {})"
by (smt RegForModule_def empty_iff empty_iff)

lemma notEmpty:
"(taking' = taking ∪ {(p,m)}) ⟶ (taking' ≠ {})"
by (smt Un_empty insert_not_empty)

end
end
```

Appendix B

ZMathLang LaTeX package

This chapter shows the ZMathLang LaTeX package which was implemented to accomadate the labels to annotate th users Z specification in ZCGa and ZDRa. This package draws the coloured boxes for the ZCGa and the boxes and labelled arrows in the ZDRa when the specification is compiled with `pdflatex`.

```
\ProvidesPackage{zmathlang}

%Packages needed for this style file
\usepackage{tcolorbox}
\usepackage{tikz}
\usepackage{varwidth}
\usepackage{zed}
\usepackage{xcolor}

%Defining the grey which would shadow out the specification but make it still visible
\definecolor{Gray}{HTML}{A0A0A0}
\definecolor{Black}{HTML}{000000}
\definecolor{White}{HTML}{FFFFFF}
\definecolor{expression}{HTML}{0FD016}
\definecolor{set}{HTML}{FF5050}
\definecolor{term}{HTML}{3A9FF1}
\definecolor{declaration}{HTML}{808080}
\definecolor{specification}{HTML}{FFE6CF}
\definecolor{text}{HTML}{C5C5C5}
\definecolor{typex}{HTML}{E4AF00}
\definecolor{boxcolor}{HTML}{000000}
\definecolor{defin}{HTML}{9999FF}

%Creating boxes to go around the schema's for the DRa
```

```
\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}
}

\newcommand\draline[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}
}

\newcommand\dratheory[3]{%
\begin{tcolorbox}[colback=white, enhanced, overlay,.
remember as=#1, scale=#2, finish={\node[] at (frame.north) {
\LARGE
\bfseries
\colorbox{Black}{\color{White}#1}};}]
{\color{Gray}\begin{varwidth}
{\dimexpr\linewidth-2\fboxsep}#3\end{varwidth}}
\end{tcolorbox}
}

%Creates the arrows (relationships) between the instances
\newcommand\initialof[2]{
\begin{tikzpicture}[overlay,remember picture
,line width=1mm,draw=black!75!black]
\draw[->] (#1.west) to[bend left] node[left, Black].
{\LARGE{initialOf}} (#2.west);
\end{tikzpicture}
}
\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}
}

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

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

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

```
%Creating coloured boxes for ZCGa weak types
\newcommand\expression[1]{
\fcolorbox{Black}{expression}{$#1$}
}

\newcommand\term[1]{
\fcolorbox{Black}{term}{$#1$}
}

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

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

\renewcommand\text[1]{
\fcolorbox{Black}{text}{$#1$}
}

\newcommand\specification[1]{
\fcolorbox{Black}{specification}{#1}
}

\newcommand\definition[1]{
\fcolorbox{Black}{defin}{$#1$}
}

\endinput
```

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.