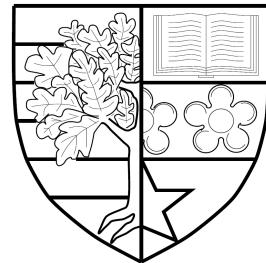


**FROM FORMAL SPECIFICATION TO FULL PROOF:
A STEPWISE METHOD**

by

Lavinia Burski



Submitted for the degree of
Doctor of Philosophy

DEPARTMENT OF COMPUTER SCIENCE
SCHOOL OF MATHEMATICAL AND COMPUTER SCIENCES
HERIOT-WATT UNIVERSITY

September 2019

The copyright in this thesis is owned by the author. Any quotation from the report or use of any of the information contained in it must acknowledge this report as the source of the quotation or information.

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.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Industries developing high integrity software are always looking for ways to make their software safer. Safety Integrity Levels (SIL) are used to define a level of risk-reduction provided by a safety-function. The highest SIL [?] which could be given to hardware or software system, as given by the International Electrotechnical Commission (IEC) [?] standard, is a SIL4. A SIL4 has a probability of failure of between 0.0001 and 0.00001 [?], and although these probabilities are very low, they are non-zero, and the upper bound of 0.000001 suggests a failure every once every 1,000,000 times on average, the outcome of which can be catastrophic.

Software testing usually takes place when the program or a prototype has been implemented. However by the time the product is fully implemented and errors are caught it is expensive to go back to the planning stage to find solutions to those bugs. Catching errors at an earlier stage of the project life cycle is more time and cost effective for the whole project team.

One way of detecting errors at an early stage 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 [?]. 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 [?] is a state machine which operates on states or arbitrary data

structures. The B-method [?] is a formal method for the development of program code from a specification in the ASM notation. Z [?] is a specification language which is used for describing computer-based systems. These are just a selection of various formal 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 [?].

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 tool-set 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 Contributions

The focus of this thesis presents a toolkit for checking various degrees of correctness for Z specifications (ZMathLang), which is an adaptation of the MathLang framework [?], [?]. Z is a formal notation to describe the functions and layout of a system (more details found in section ?? of this thesis). ZMathLang is a collection of tools which assist the user in checking the correctness of formal specifications, and also assist users by partially automating the translation of the specification into a theorem prover. 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 specification into Isabelle. We summaries 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 (described in chapter ?? and entire thesis).

2. Built a collection of tools to enable a step by step approach for Z specifications (includes ZCGa implementation in chapter ??, ZDRA implementation in chapter ?? and further aspects in chapters ??, ?? and ??).
3. Formalised and proved properties on a number of examples of Z specifications (described in chapters ??, ?? and in [?]).
4. Demonstrated and evaluated the performance of the tool set on a convincing set of examples (described in chapter ?? and ??).

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

Translating specifications into theorem provers has been 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. The ZMathLang tool-set 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 along 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 below 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 example can be checked (see Z Core Grammatical aspect (ZCGa)

chapter ??) without it needing to be fully formalised. The documents rhetorical correctness (see Z Document Rhetorical aspect (ZDRA) chapter ??) 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. The ZMathLang framework has stemmed from the MathLang framework for mathematics (MathLang) framework [?] and has been adapted to work for Z specifications.

1.1.2 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 specification 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 stakeholders of the project and can be used to describe to clients, developers, managers etc.

1.1.3 Formalised and proved properties on a number of examples of Z specifications.

A third contribution given in this thesis is that we have formalised and translated Z specifications into the theorem prover Isabelle and checked the sanity of these specifications. The sanity properties (described in the next section) were automatically generated by the ZMathLang toolkit and we have manually proved the properties in Isabelle.

In some examples we have added extra properties (such as the birthdayBook example) to show what other properties may be proved in Isabelle. Although these properties have not been automatically generated, they may be of some importance to some users. Other properties which could be manually inputted are described in section ??.

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

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 ?? in chapter ??). 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.

1.1.4 Demonstrated and evaluated the performance of the tool set 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 [?]
- Clubstate [?]
- ModuleReg [?]
- TelephoneBook
- VideoShop [?]
- A specification which fails ZCGa
- A specification which fails ZDRa
- Autopilot (A semi formal Specification) [?]
- The ZCGa type checker

- Vending Machine [?]
- Clubstate2 [?]
- ProjectAlloc [?]
- Timetable [?]
- Steamboiler [?]

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 annotations needed for each specification.

1.2 How far does the automation go?

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

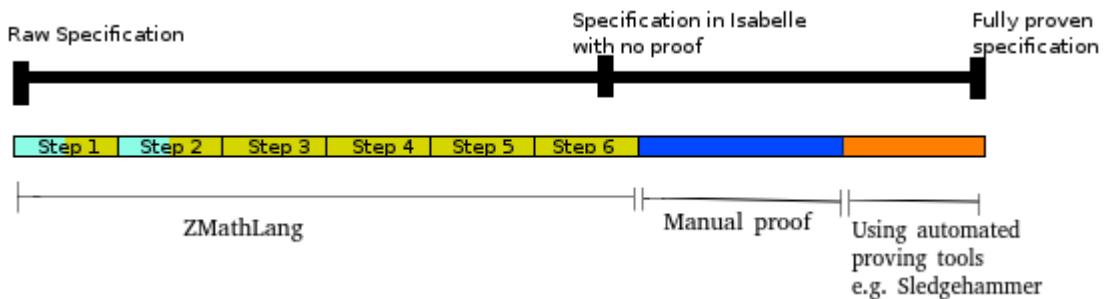


Figure 1.1: How far can one automate a specification proof.

In figure ?? we show how far the user can get with automation and how much work is still needed to get the full proof ¹. ZMathLang requires user input for the first two steps (ZCGa and ZDRa) however the rest up to step 6 is automated.

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

We have created the ZMathLang toolkit which assist the user from the specification to full proof however there is also ongoing research on proving properties from the theorem prover end. We have highlighted that ZMathLang only gets the user so far in their proof however they are free to use external automated theorem provers such as sledgehammer in completing their specification proof if they so wish.

Even external automated theorem provers have their limitations. For example, the user can use the Isabelle tool ‘*sledgehammer*’ to assist in solving proofs, but not all can be solved by this technique. The sledgehammer documentation advises to call ‘*auto*’ or ‘*safe*’ followed by ‘*simp_all*’ before invoking sledgehammer. Depending

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

on the complexity of ones proof, these sometimes may prove the users properties on their own, other times it may not and the user will still need to invoke sledgehammer to reach their goal. Sledgehammer itself is a tool that applies Satisfiability Modulo Theories (SMT) solvers on the current goal e.g. Vampire[?], SPASS [?] and E [?]. We use sledghammer as a collective, to describe all the SMT solvers it covers [?]. However, there still doesn't exist an automated proving tool which covers **all** proving techniques. Therefore some user input will be required for more complex proofs.

1.3 Outline

In chapter ?? we begin describing the origins of 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 ?? 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 [?], is thoroughly described and details of how the tool can be used are given.

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

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

Chapter ?? 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 ?? 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

Formal methods are a specific type of mathematical notation which is based on the techniques of the specification, verification and development of software and hardware systems [?]. Since our thesis presents ZMathLang we go right to the beginning of the framework, to describe how mathematical notation came about. Then we describe the original MathLang framework (the framework which ZMathLang is an adaptation of) and then give the reader an idea of other formal methods and languages. In the next section we wish to describe what is the language of Z and give more details of it's syntax and semantics. We then highlight other proving techniques which have been done for maths, formal methods and Z.

2.1 Mathematical Notations

Computer science (and thus computer systems) have evolved from basic mathematics. We can say that formal specification writers are practising mathematicians as they write system specifications in a formal manner. Therefore we must start right at the beginning at the foundation of mathematical notation.

2.1.1 Right from the beginning

The relationship between mathematical reasoning and practising mathematicians started out early on during the ancient Greeks where logic was already being studied. Reasoning in logic was used for just about anything not just mathematics such as

law, medicine and farming. This very early form of mathematics made very famous discoveries such as Aristotles logic [?], Euclid's geometry [?] and Leibniz Calculus [?].

Further on in the 1800's, Frege wrote *Die Grundlagen der Arithmetik* [?] and other works where he noted that mathematics is a branch of logic. In this works, he began building a solid foundation for mathematics. This early foundation along with Cantors set theory [?] was argued to be inconstant and thus Russel found a paradox in this work.

In the late 19th century and beginning of 20th century, Russell & Whitehead [?] 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 derive 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, Bourbaki¹ wrote a series of books beginning n the 1935's with the aim of grounding mathematics. Their main works is included in the Elements of Mathematics series [?] 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.

Adding to Russell's work, Zermelo introduced an acclimatization of set theory which was later extended by Frankel and Skolem to form ZF set theory [?]. This new theory is what we will later see the Z notation is based on and the notation this thesis checks the correctness of.

2.1.2 Computerisation of Maths and Proof Systems

In the 21st century, a great area of research is how use, store and support this mathematical knowledge. Since automation has become more and more used, mathematicians have looked into ways in which they can use computers to reason about and provide services to mathematics. This would include all areas of mathematics,

¹A name given to a collective of mathematicians

such as logic, mechanics and software specifications. Mathematical knowledge can be represented in lots of different ways including the following:

- One can typeset mathematics into a computer using a system such as L^AT_EX [?]. These systems can edit and format mathematical knowledge so that it can be stored or printed. These systems provide good visual appearance and thus can be used for storing and archiving ones documents. Even Z specifications, have their own package for L^AT_EX and thus the structure of the specification can be represented both formally and informally. However, it is difficult to represent the logical structure of mathematical formulas and the logical structure of mathematics is embedded in natural language. Therefore, there isn't a lot of support for checking the correctness of general mathematics in the system (Z specifications on their own can be checked but this is discussed further in section ??).
- Systems such as proof assistants (e.g. Isabelle [?], Coq [?] and ProofPower-Z [?]) and automated theorem provers (e.g. Boyer-Moore, Otter) are jointly called proof systems. Proof systems each provide a formal language for writing mathematics. Early work on proof systems was done by De Bruijn when he worked on the AutoMath [?] project. AutoMath (automating mathematics) was the first attempt to digitize and formally prove mathematics which was assisted 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 to this work, there has been many other proof systems created to implement and check mathematics for total correctness. It is possible to access the semantics of mathematics in these systems. However, with these proof systems, a user must choose a specific proof system and all of these have their advantages and disadvantages. Also, each of these proof systems also take quite some time to learn. There is much documentation on some of these systems (e.g. Isabelle) and some are very well supported. But this in turn

can be a downfall, as there is so much documentation, it is difficult to know how much one must learn and where to start. The best way of learning one of these system is from someone who already is an expert in the chosen proof system. A lot of the proof systems use proof tactics to constructs proofs and make them smaller, however to prove certain properties in a proof system, one can use various tactics to get to the same goal and may sometimes be difficult to find which tactic is best to use.

With these disadvantages many academic and industrial mathematicians do not generally use the mathematics written in the language of the proof system and usually are not willing to spend the time to check the correctness of their own work in this system.

- There also exists semantical oriented document representations like OpenMath [?] and OMDoc [?]. Some aspects of the semantics of the mathematics can be represented in these type systems. However, when using these systems it is difficult to control the presentation and therefore a typesetting system is more likely to be used. Like the typesetting system, OMDoc has also have difficult reading the logical structure of mathematics embedded in the natural language of the text. Systems like OMDoc can associate symbolic formulas with chunks of natural language text, however these chunks can not be seen by the computer and thus can not be checked if it is correct.

Another disadvantage is that although there is support for the semantics of the mathematics, these systems can not support the semantics in terms of logical foundations for mathematics (unlike proof systems).

2.1.3 Conclusion

In summary the MathLang framework has been developed to be used as a bridge between the categories mentioned above as a way to represent and automatically check mathematical knowledge. Since the Z notation has stemmed from the origins of mathematics and industry is starting to use formal methods in there system

development we have chosen to adapt the MathLang framework to accommodate Z specification and have developed a set of tools to do so.

2.2 MathLang for mathematics

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

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 not 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 [?] and his proof checking system Automath.

2.2.1 Overview and Goals

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.

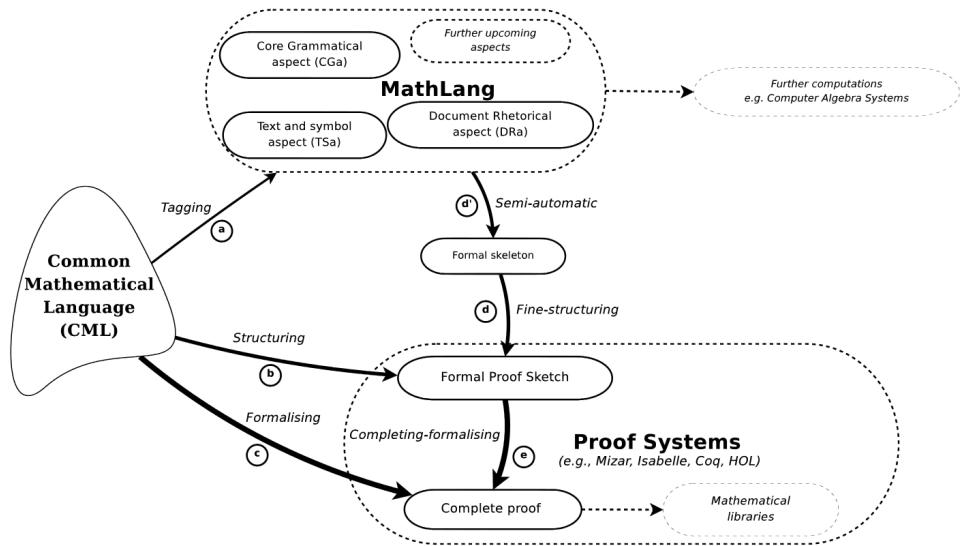


Figure 2.1: The MathLang approach to computerisation/formalisation [?]

Figure ?? 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.

The initial design of MathLang was based on ideas from Weak Type Theory (WTT) [?] which then inspired de Bruijns Mathematical Vernacular [?]. WTT defines terminology to describe a *book* which is a sequence of *lines*, each of which is a pair of a *sentence* (a *statement* or a *definition*) and a *context* of facts (*declarations* or *statements*) assumed in the *sentence*. A *definition* can introduce a name whose scope is associated with a meaning throughout the rest of the book. A *declaration* in the *context* can introduce a name whose scope is the current line. The *preface* gives names which can be reference throughout the entire document. Within *declarations*, *definitions* and *statements* are *phrases* which can be built from the basic elements; *terms*, *sets*, *adjectives* and *nouns*. WTT uses a weak type system using the following types **noun**, **set**, **adjective**, **statement**, **definition**, **context** and **book** to check basic well-formedness conditions. The idea's of WTT are useful in some instances

however it does not deal with proofs or logical correctness of the documents. There is no way to group statements together and identify their roles within the document such as what is a theorem, what is a section, what is a chapter etc. It also doesn't provide a way to use some concepts used in a document within another.

We use the following standard meta-symbols for the categories mentioned above:

level	category	symbol	representative
atomic level	<i>variables</i>	V	x
	<i>constants</i>	C	c
	<i>binders</i>	B	b
phrase level	<i>terms</i>	t	t
	<i>sets</i>	σ	s
	<i>nouns</i>	\mathcal{V}	n
	<i>adjectives</i>	α	a
sentence level	<i>statements</i>	\mathcal{S}	S
	<i>definitions</i>	\mathcal{D}	D
discourse level	<i>contexts</i>	Γ	Γ
	<i>lines</i>	l	l
	<i>books</i>	B	B
category	symbol	representative	
<i>expressions</i>	\mathcal{E}	E	
<i>parameters</i>	\mathcal{P}	P	
<i>typings</i>	\mathcal{T}	T	
<i>declarations</i>	\mathcal{Z}	Z	

Variables can range of terms, sets or statements.

Parameters \mathcal{P} are either terms, sets or statements. A *constant* is always followed by a parameter list and constants can be either for terms, nouns, statements, sets or adjectives. For example:

The constant '+' takes 2 terms as parameters and gives back a term, therefore we can have:

$$+(3, 2)$$

meaning the constant `+` takes the parameters 3 and 2 and in this case would give back 5.

Binders can be giving terms, adjectives, nouns, sets or statements. For example a binder giving a set would be the symbol ‘ \cup ’ or a binder giving a statement would be the symbol ‘ \forall ’.

A *definition* introduces new constants. We can have phrase definitions and statement definitions.

A *context* is a list of declarations and statements. A declaration in a context introduces variables which are then used in expressions. A context can be empty or non empty.

Lines contain either a statement or a definition which is relative to a context. A *book* is a list of lines. A simple book could be empty and not contain any lines or it can contain 1 or more lines.

A detailed explanation on each of the categories along with examples can be found in [?].

2.2.2 CGa

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 [?].

The CGa for mathematical documents is a formal language originally created by ideas by WTT and Mathematical Vernacular [?] and then adapted to suit mathematical texts by various PhD, Master and Dissertation students.

The smallest constructs of CGa are *step* and *expression*. Tasks handled by books, prefaces, lines, declarations, definitions, and statements are represented as steps in

CGa. A step can be a *block* which is a sequence of steps. A step can also be a definition, declaration or expression.

For example:

‘Given that \mathfrak{M} is a set, y and x are natural numbers, and x belongs to \mathfrak{M} , it holds

that $x + y = y + x$ ’.

This statement encoded in CGa would be the following:

$$\begin{aligned} \{M : set; y : naturalnumber; x : naturalnumber; \in (x, M)\} \\ \triangleright= (+ (x, y), +(y, x)) \end{aligned}$$

CGa uses grammatical *categories* (also called *types*) to make explicit the grammatical role played by the elements of a mathematical text in a document. All these described categories and the rules for the grammar typing can be found in [?]. CGa also uses colour coding for these categories. They are:

`term` , `set` , `noun` , `adjective` , `expression` , `definition` , `declaration` ,
`step` and `context`

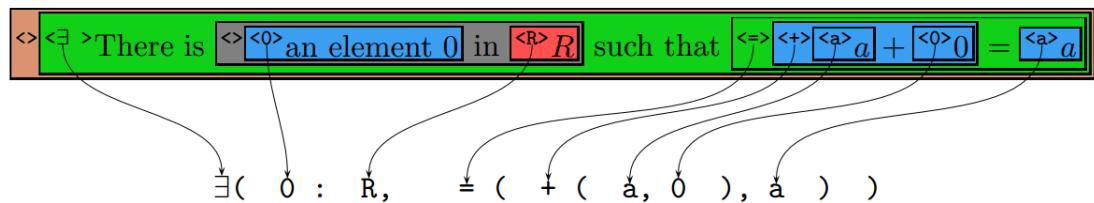


Figure 2.2: Example of CGa encoding of CML text, taken from [?].

CGa types are powerful and do not allow higher order types. The goal of the CGa is not to endure full correctness but to check whether the reasoning parts of the document are coherent and built sensibly.

The design of the CGa is credited to Kamareddine, Maarek and Wells [?]. The implementation of CGa is due to Maarek [?].

2.2.3 DRa

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 DocBook², Text Encoding Initiative³, OMDoc⁴) 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 [?]. 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 circular reasoning and poorly-supported theorems.

The annotation process for the DRa can be done before, after or with the annotations for CGa and TSa. Table ?? shows the roles and relations which can be used in annotating a document with DRa.

Roles	Relations
lemma	justifies
proof	uses
corollary	subpartOf
definition	
claim	
case	

Table 2.1: A table showing DRa roles and relations.

Annotations in DRa follow three steps:

1. The user wraps chunks of text with boxed. Each box is assigned a unique

²<http://www.docbook.org>

³<http://www.tei-c.org/index.xml>

⁴<http://www.omdoc.org>

name, this avoids any confusion when stating relations between the boxes.

2. To each unique box, the user then assigns a structural or mathematical rhetorical role which this box may play. He uses RDF triplets to assign a role to each box.
3. The user then describes the relations between each of the chunks of text he annotated before.

RDF triples [?] are used to represents the relationships between the annotated boxes. The triples are represented as *subject-predicate-object* triple, where *predicate* denotes the relationship. The order of *subject* and *object* is important as it denotes how the dependency graph is formed. An example of RDF triples for the ZDRa is:

Assigned Rhetorical Roles	Relations
(A, hasMathematicalRole, lemma)	(B, justifies, A)

The DRa performs three basic checks:

1. Check if each of the mathematical and rhetorical roles and relations which are used have been defined.
2. Check that each name of a node is unique.
3. Check that each relation source and target are valid DRa names.

The DRa also automatically produces a dependency graph.

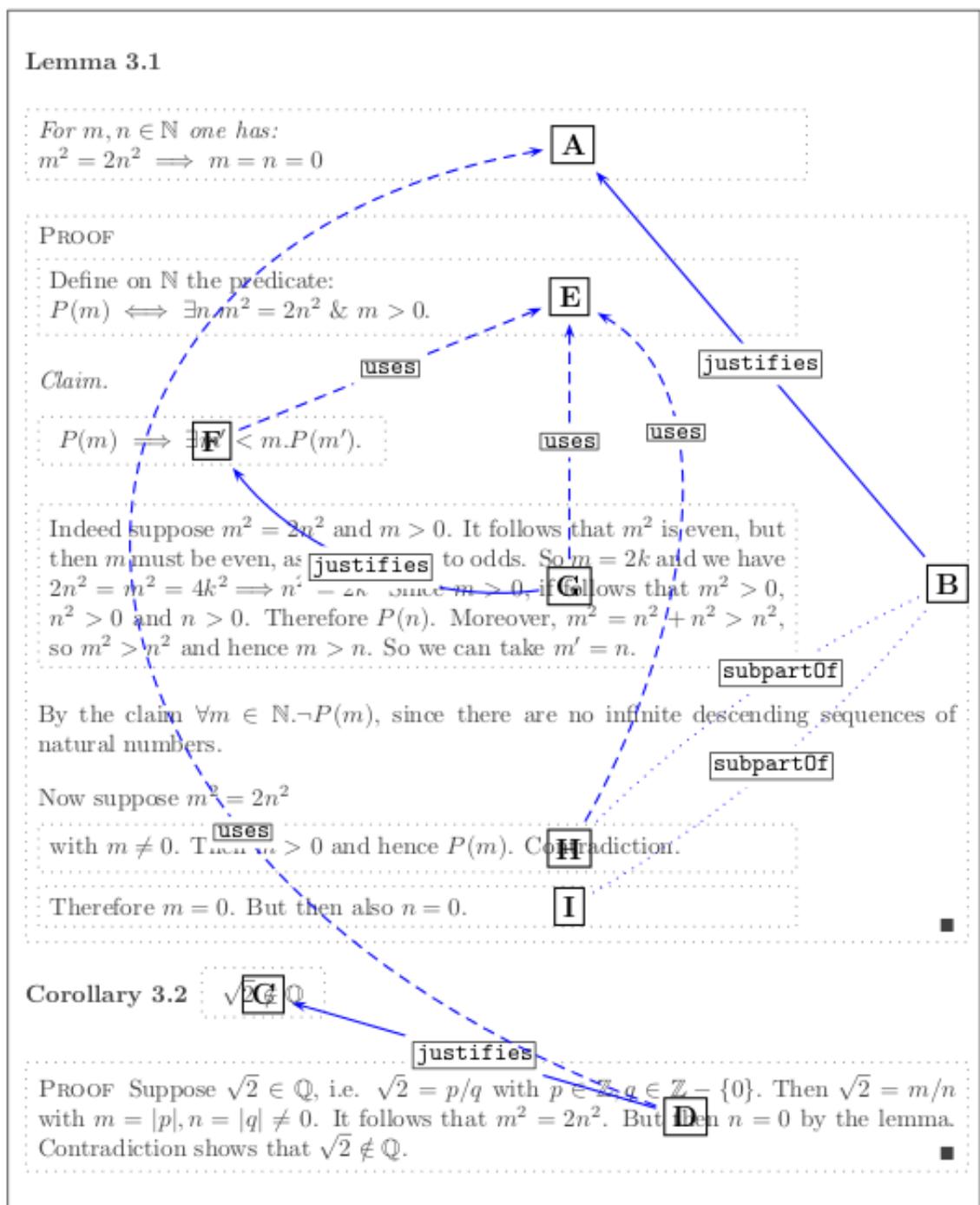


Figure 2.3: Wrapping/naming chunks of text and marking relationships in DRa [?].

2.2.3.1 Dependency Graph

The dependency graph is a directed label graph which contains all annotated DRa nodes of a mathematical text and all the relations. The names of the instances become the nodes of the graph and the relations become the named directed vertices of the graph [?].

Since each DRa annotation itself is a step, DRa annotations can be nested.

An example of a dependency graph is shown in figure ??:

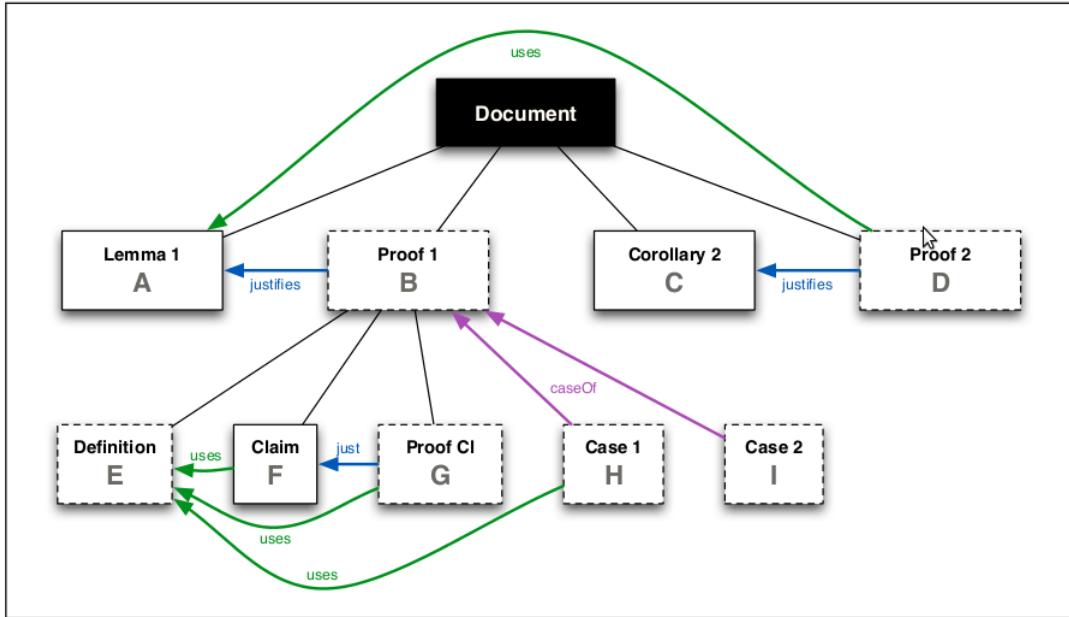


Figure 2.4: The Dependency Graph for the Pythagoras example, taken from [?].

2.2.3.2 GoTo Graph

The textual order between two nodes within a tree can be used to examine the structure of the DRa. The textual order is a modification of the logical precedence presented in the dependency graph. The textual order expresses the dependencies between parts of the text and is necessary for the generation of a proof skeleton. For example if a node A uses node B than logically B should be written before A. The GoTo graph is the same as the dependency graph but rearranged to follow the correct textual order of the nodes and vertices. The GoTo graph is produced for the following reasons:

1. The automatic checking of the GoTo can reveal errors in the document such as loops in structure of the document.
2. The GoTo is used to produce a proof skeleton for a certain prover.

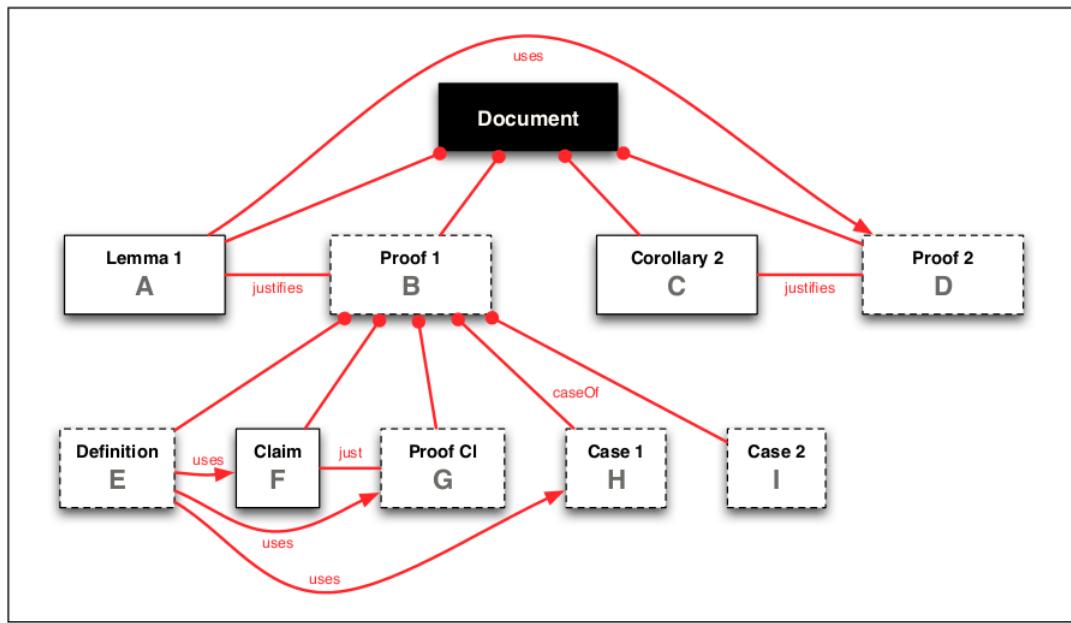
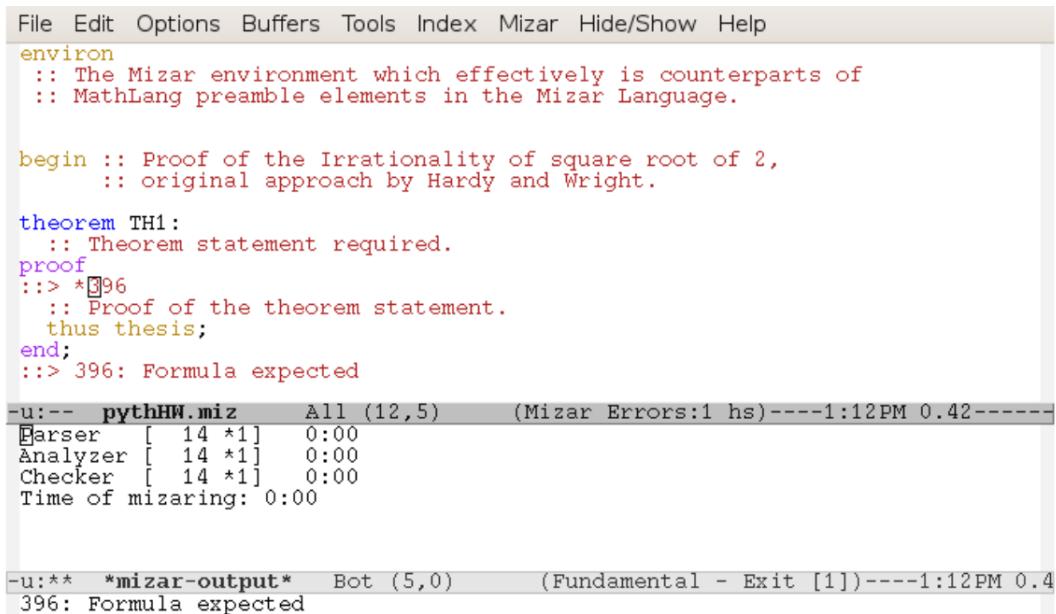


Figure 2.5: Example of the GoTO of a document, taken from [?].

2.2.4 Mathematical skeletons

There has been some work on computerising mathematical texts without fully formalising or proof checking them. The computerisations are no detailed to check for full total correctness but are used as *skeletons* in the full formalisation [?]. However, in order to produce these skeletons the user still needs to be an expert in the theorem proving language they wish to check their mathematical document with. Once a document has been annotated with MathLang, the proof skeleton for a theorem prover could be generated from the annotations. A formal proof sketch is a document written in the syntax of a certain theorem prover, with the allowance that it may be checked by the system with errors however they should only be justification errors.

Retel [?] describes a Mizar document skeleton which has been generated from a MathLang document. The skeleton uses the annotations from DRa and inputs the document in Mizar syntax along the names used in DRa. The original context of the mathematical document has not yet been input into the Mizar formal proof skeleton.



```

File Edit Options Buffers Tools Index Mizar Hide/Show Help
environ
  :: The Mizar environment which effectively is counterparts of
  :: MathLang preamble elements in the Mizar Language.

begin :: Proof of the Irrationality of square root of 2,
  :: original approach by Hardy and Wright.

theorem TH1:
  :: Theorem statement required.
proof
::> *396
  :: Proof of the theorem statement.
  thus thesis;
end;
::> 396: Formula expected

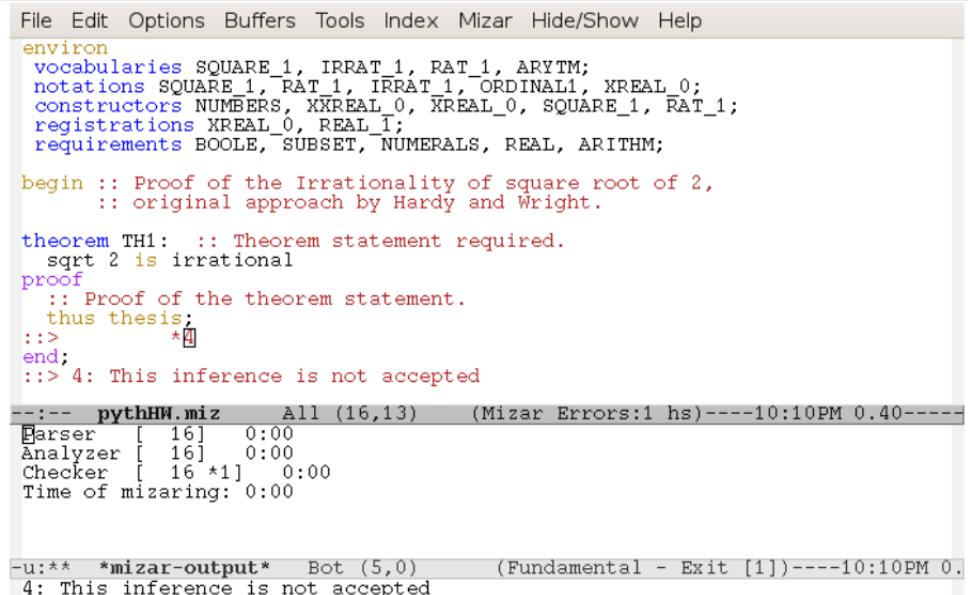
-u--- pythHW.miz   All (12,5)      (Mizar Errors:1 hs)---1:12PM 0.42-----
Parser [ 14 *1] 0:00
Analyzer [ 14 *1] 0:00
Checker [ 14 *1] 0:00
Time of mizaring: 0:00

-u:** *mizar-output* Bot (5,0)      (Fundamental - Exit [1])---1:12PM 0.4
396: Formula expected

```

Figure 2.6: The Mizar proof document skeleton transformed from the MathLang computerised document [?].

The next part is to fill in the skeleton using all information and context of the original mathematical document, this may include the names of the theorems, statements, definitions etc. which were in the original mathematical text.



```

File Edit Options Buffers Tools Index Mizar Hide/Show Help
environ
  vocabularies SQUARE_1, IRRAT_1, RAT_1, ARYTM;
  notations SQUARE_1, RAT_1, IRRAT_1, ORDINAL1, XREAL_0;
  constructors NUMBERS, XXREAL_0, XREAL_0, SQUARE_1, RAT_1;
  registrations XREAL_0, REAL_1;
  requirements BOOLE, SUBSET, NUMERALS, REAL, ARITHM;

begin :: Proof of the Irrationality of square root of 2,
  :: original approach by Hardy and Wright.

theorem TH1: :: Theorem statement required.
  sqrt 2 is irrational
proof
  :: Proof of the theorem statement.
  thus thesis;
::> *
end;
::> 4: This inference is not accepted

--:- pythHW.miz   All (16,13)      (Mizar Errors:1 hs)---10:10PM 0.40-----
Parser [ 16] 0:00
Analyzer [ 16] 0:00
Checker [ 16 *1] 0:00
Time of mizaring: 0:00

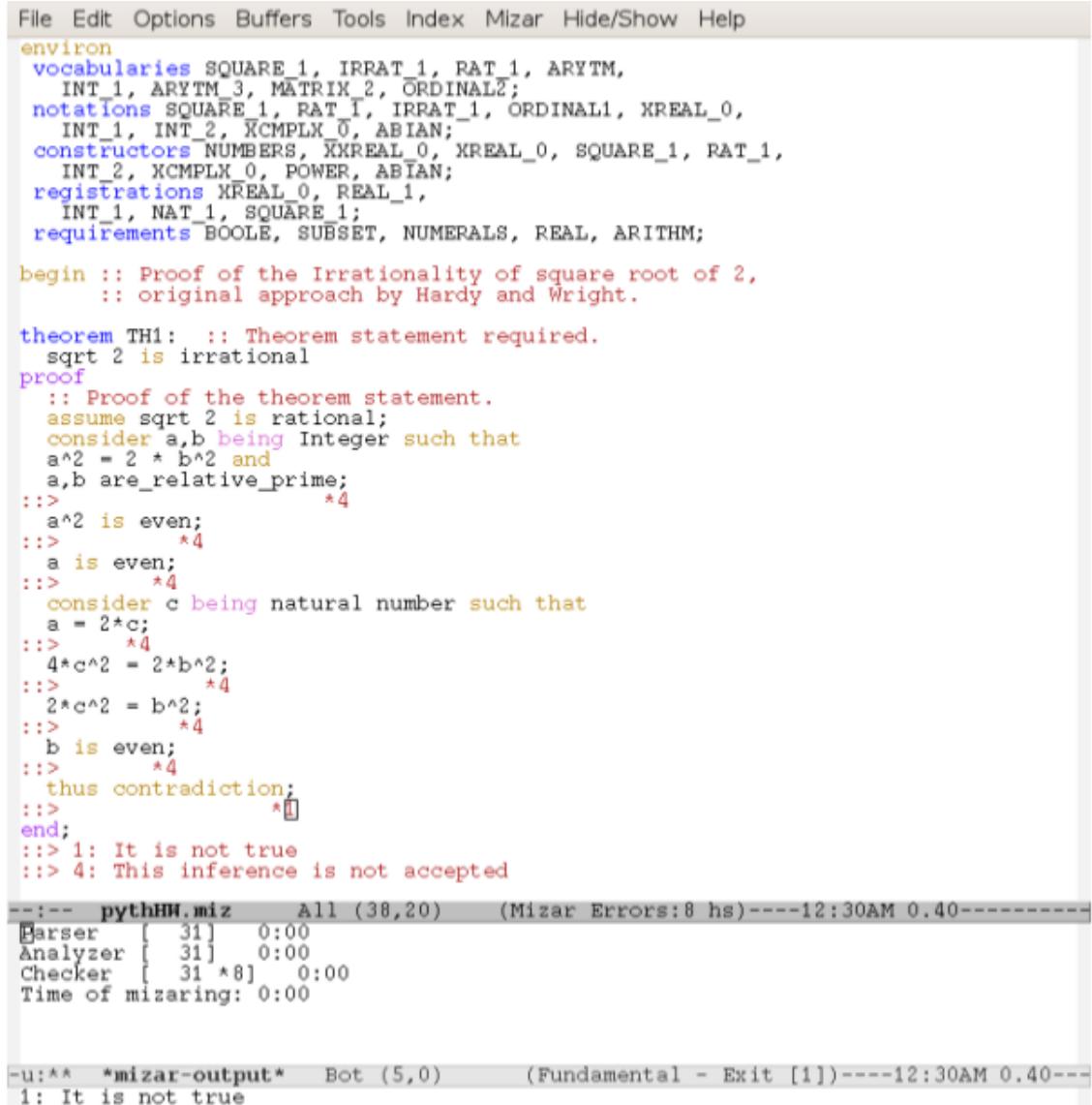
-u:** *mizar-output* Bot (5,0)      (Fundamental - Exit [1])---10:10PM 0.
4: This inference is not accepted

```

Figure 2.7: The Mizar FPS environment built for the example from Figure ???. The Mizar system responds with only one error stating that the theorem is not sufficiently proved, taken from [?].

Figure ?? still shows that there is still some work to do as the ‘*inference is not accepted*’. The user would then need to finish off the proof manually (shown

in figure ??), however a big chunk of the work has already been done within the theorem prover.



```

File Edit Options Buffers Tools Index Mizar Hide/Show Help
environ
  vocabularies SQUARE_1, IRRAT_1, RAT_1, ARYTM,
    INT_1, ARYTM_3, MATRIX_2, ORDINALZ;
  notations SQUARE_1, RAT_1, IRRAT_1, ORDINAL1, XREAL_0,
    INT_1, INT_2, XCMPLX_0, ABIAN;
  constructors NUMBERS, XXREAL_0, XREAL_0, SQUARE_1, RAT_1,
    INT_2, XCMPLX_0, POWER, ABIAN;
  registrations XREAL_0, REAL_1,
    INT_1, NAT_1, SQUARE_1;
  requirements BOOLE, SUBSET, NUMERALS, REAL, ARITHM;

begin :: Proof of the Irrationality of square root of 2,
  :: original approach by Hardy and Wright.

theorem TH1: :: Theorem statement required.
  sqrt 2 is irrational
proof
  :: Proof of the theorem statement.
  assume sqrt 2 is rational;
  consider a,b being Integer such that
  a^2 = 2 * b^2 and
  a,b are_relative_prime;
  ::> a^2 is even; *4
  ::> a is even; *4
  ::> consider c being natural number such that
  a = 2*c; *4
  ::> 4*c^2 = 2*b^2; *4
  ::> 2*c^2 = b^2; *4
  ::> b is even; *4
  ::> thus contradiction; *1
end;
::> 1: It is not true
::> 4: This inference is not accepted
--:-- pythHW.miz   All (38,20)  (Mizar Errors: 8 hs)---12:30AM 0.40---
Parser [ 31] 0:00
Analyzer [ 31] 0:00
Checker [ 31 *8] 0:00
Time of mizaring: 0:00

-u:** *mizar-output* Bot (5,0)      (Fundamental - Exit [1])---12:30AM 0.40--
1: It is not true

```

Figure 2.8: A filled in Mizar proof built from Figure ?? taken from [?].

2.2.5 TSa

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.

We do not describe the TSa in detail since we do not use the TSa in this thesis but it is important to know that it exists.

2.2.6 A fully worked example in MathLang

This section describes a fully worked example of translating a mathematical text into the theorem prover Isabelle using the MathLang framework, taken from [?].

2.2.6.1 The original text

This text was taken from an undergraduate textbook [?], chapter 12, entitled ‘Introduction to Rings’.

Definition 2.2.1. (*Ring*). A ring R is a non-empty set with two binary operations, addition (denoted by $a + b$) and multiplication (denoted by ab), such that for all a, b, c in R :

1. $a + b = b + a$.
2. $(a + b) + c = a + (b + c)$.
3. There is an additive identity 0 . That is, there is an element 0 in R such that $a + 0 = a$ for all a in R .
4. There is an element a in R such that $a + (a) = 0$.
5. $a(bc) = (ab)c$.
6. $a(b + c) = ab + ac$ and $(b + c)a = ba + ca$. content...

Our first theorem shows how the operations of addition and multiplication intertwine.

Theorem 2.2.1. (*Rules of Multiplication*). Let a, b , and c belong to a ring R . Then

1. $a0 = 0a = 0$.
2. $a(b) = (a)b = (ab)$.

Proof. Consider rule 1. Clearly,

$$0 + a0 = a0 = a(0 + 0) = a0 + a0.$$

So, by cancellation, $0=a0$. Similarly, $0a=0$. To prove rule 2, we observe that $a(b) + ab = a(b + b) = a0 = 0$. So, adding (ab) to both sides yields $a(b) = (ab)$. The remainder of rule 2 is done analogously. \square

2.2.6.2 CGa annotated text

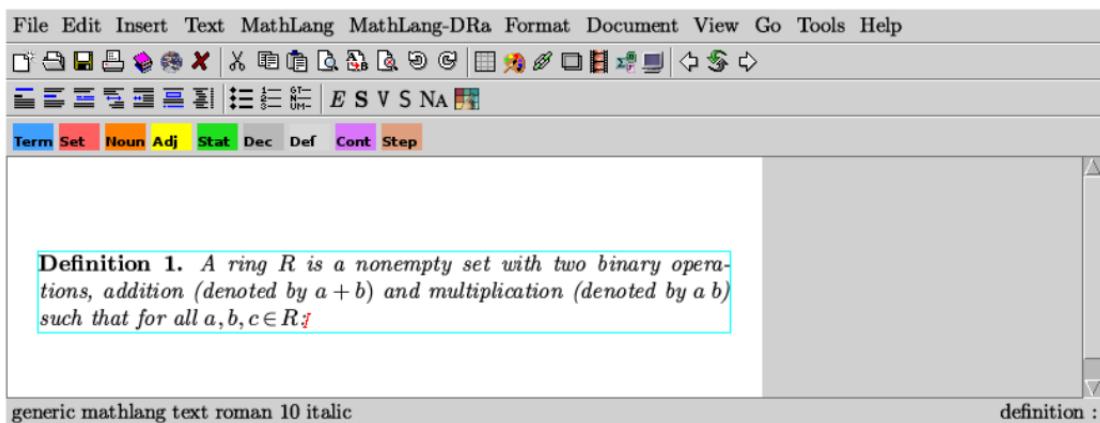


Figure 2.9: Typing the beginning of a document into TexMacs.

The user then annotates this text using ‘*TeXMacs*’ shown in figure ???. These annotations would be made at sentence level and it is common for single words and individual variables to be annotated.

definition

ring

Definition 1. A $\text{ring } R$ is a non-empty set with two binary operations, addition (denoted by $\text{plus } a + b$) and multiplication (denoted by $\text{times } a \cdot b$), such that for all a, b, c in R :

1. $\text{equal}(\text{plus}[a] + [b], \text{plus}[b] + [a])$.
2. $\text{equal}(\text{plus}([\text{plus}[a] + b] + c), \text{plus}[a] + ([\text{plus}[b] + c]))$.
3. There is an additive identity 0 . That is, there is an element $\text{zero } 0$ in R such that $\text{equal}(\text{plus}[a] + \text{zero } 0, a)$ for all a in R .
4. There is an element $\text{negative } -a$ in R such that $\text{equal}(\text{plus}[a] + (\text{negative } -a), \text{zero } 0)$.
5. $\text{equal}(\text{times}[a] (\text{times}[b] \cdot c), \text{times}([\text{times}[a] b] \cdot c))$.
6. $\text{equal}(\text{times}[a] (\text{plus}[b] + c), \text{plus}(\text{times}[a] b) + \text{times}[a] c)$ and $\text{equal}(\text{times}([\text{plus}[b] + c] a), \text{plus}(\text{times}[b] a) + \text{times}[c] a)$.

theorem

Theorem 2. ring

1. $\text{rule1} \quad \text{and equal } r \cdot \text{times}[x.a] \text{ zero } 0 = \text{zero } r \cdot \text{times}[x \cdot \text{zero } 0] \cdot a \text{ equal } = r \cdot \text{zero } 0$.
2. $\text{rule2} \quad \text{and equal } r \cdot \text{times}[x.a] (\text{r.negative } - [x.b]) = \text{rule1 } r \cdot \text{times}([x \cdot \text{negative } - [x.a]] a) \cdot b \text{ equal loop } = r \cdot \text{negative } - r \cdot \text{times}[x.a] \cdot r.b$.

proof

Proof. ring

rule1 Consider rule 1.

Clearly,

$$\text{equal}(r \cdot \text{plus}[x \cdot \text{zero } 0] + r \cdot \text{times}[x.a] \text{ zero } 0) = \text{shared } r \cdot \text{times}[x.a] \text{ zero } 0 \text{ equal } = \text{shared } r \cdot \text{times}[x.a] (\text{r.plus } x \cdot \text{zero } 0 + r \cdot \text{zero } 0) \text{ equal } = r \cdot \text{plus}[\text{shared } r \cdot \text{times}[x.a] \text{ zero } 0] + r \cdot \text{times}[x.a] \text{ zero } 0. \quad (1)$$

So, by cancellation, $\text{equal } r \cdot \text{zero } 0 = r \cdot \text{times}[x.a] \text{ zero } 0$. Similarly, $\text{equal } r \cdot \text{times}[x \cdot \text{zero } 0] \cdot a = r \cdot \text{zero } 0$.

To prove rule 2, we observe that $\text{equal}(r \cdot \text{plus}[r \cdot \text{times}[x.a] (\text{r.negative } - [x.b])] + r \cdot \text{times}[x.a] r.b) = \text{shared } r \cdot \text{times}[x.a] (\text{r.plus } r \cdot \text{negative } - [x.b] + r.b) \text{ equal } = \text{shared } r \cdot \text{times}[x.a] \text{ zero } 0 \text{ equal } = r \cdot \text{zero } 0$. Adding $- (ab)$ to both sides yields $\text{equal } r \cdot \text{times}[x.a] (\text{r.negative } - [x.b]) = r \cdot \text{negative } - (r \cdot \text{times}[x.a] r.b)$. The remainder of rule 2 is done analogously. \square

Figure 2.10: Completed CGa annotation of ring theory text.

2.2.6.3 DRa annotated text

The text is annotated with DRa relationships and instances which can find errors such as loops within the reasoning and poorly-supported theorems. Figure ?? shows the original ring theory text (left) and the ring theory text annotated with DRa (right).

Definition 9.1.

A ring R is a nonempty set with two binary operations, addition (denoted by $a + b$) and multiplication (denoted by ab), such that for all a, b, c in R :

1. $a + b = b + a$.
2. $(a + b) + c = a + (b + c)$.
3. There is an additive identity 0 . That is, there is an element 0 in R such that $a + 0 = a$ for all a in R .
4. There is an element $-a$ in R such that $a + (-a) = 0$.
5. $a(bc) = (ab)c$.
6. $a(b + c) = ab + ac$ and $(b + c)a = ba + ca$.

Theorem 1.

Rules of Multiplication Let a, b , and c belong to a ring R . Then

1. $a0 = 0a = 0$.
2. $a(-b) = (-a)b = -(ab)$.

Proof.

Consider rule 1. Clearly,

$$0 + a0 = a0 = a(0 + 0) = a0 + a0.$$

So, by cancellation, $0 = a0$. Similarly, $0a = 0$.

To prove rule 2, we observe that $a(-b) + ab = a(-b + b) = a0 = 0$. So, adding $-(ab)$ to both sides yields $a(-b) = -(ab)$. The remainder of rule 2 is done analogously.

Definition 9.1.

A ring R is a nonempty set with two binary operations, addition (denoted by $a + b$) and multiplication (denoted by ab), such that for all a, b, c in R :

1. $a + b = b + a$.
2. $(a + b) + c = a + (b + c)$. **D1**
3. There is an additive identity 0 . That is, there is an element 0 in R such that $a + 0 = a$ for all a in R .
4. There is an element $-a$ in R such that $a + (-a) = 0$. **subpartOf**
5. $a(bc) = (ab)c$. **uses**
6. $a(b + c) = ab + ac$ and $(b + c)a = ba + ca$. **uses**

Theorem 1
uses

Rules of Multiplication Let a, b , and c belong to a ring R . Then

1. $a0 = 0a = 0$. **TH2**

2. $a(-b) = (-a)b = -(ab)$. **subpartOf**

Proof.

Consider rule 1. Clearly,

$$0 + a0 = a0 = a(0 + 0) = a0 + a0.$$

P2a

So, by cancellation, $0 = a0$. Similarly, $0a = 0$.

subpartOf

To prove rule 2, we observe that $a(-b) + ab = a(-b + b) = a0 = 0$. So, adding $-(ab)$ to both sides yields $a(-b) = -(ab)$. The remainder of rule 2 is done analogously.

P2b
C12

Figure 2.11: Comparison of original common mathematical language text with a dependency graph.

The dependency graphs and goto graphs are automatically generated from the DRa annotated document shown in figure ??.

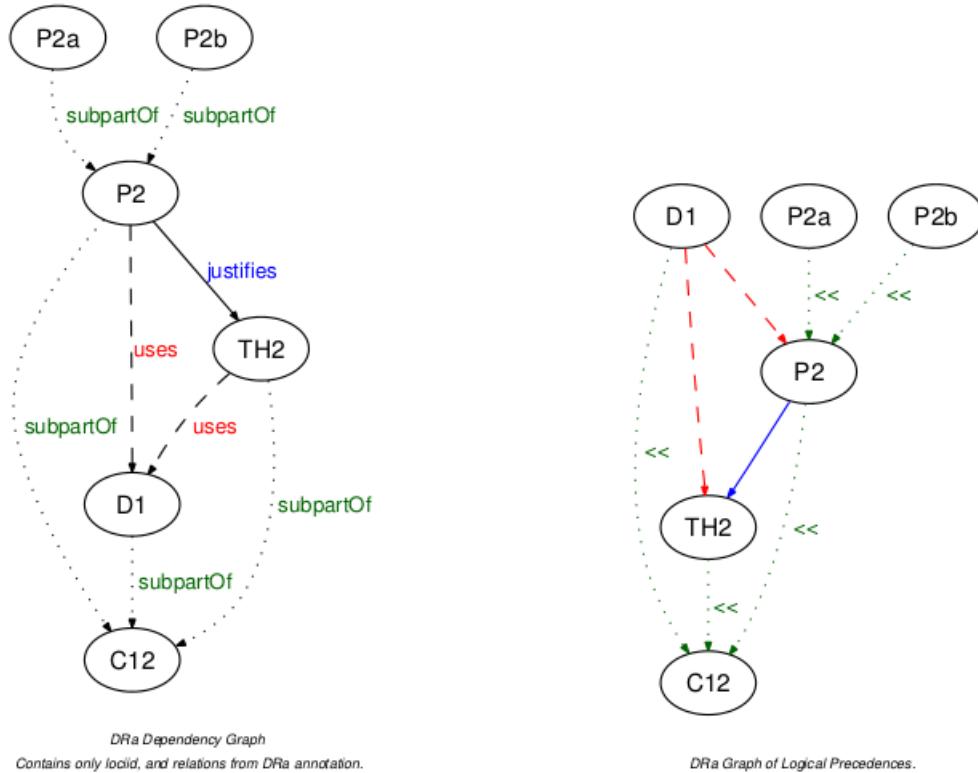


Figure 2.12: Two graphs automatically generated from DRa annotations: The dependency graph (DG) and the graph of logical precedence (GoTo).

2.2.6.4 Translating CGa and DRa into Isabelle syntax

In this thesis we refer to this part as the filled in Isabelle skeleton or halfbaked proof. This document is now legal Isabelle syntax, but is not formally correct. There still may be gaps in the reasoning or particular proofs which need completing. However, this file may now be worked on by an Isabelle expert to finish off the final proof. The Isar command `sorry` tells Isabelle to skip a proof-in-progress and treat the goal under consideration to be proved.

Note: Translating just the DRa annotations gives the Isabelle Proof Skeleton and translating the CGa **and** DRa annotations gives the halfbaked proof.

```
5  locale ring =
6    fixes R :: "'r set"
7    assumes nonempty: "R \<noteq> {}"
8    fixes plus :: "'r, 'r] => 'r"
9    fixes times :: "'r, 'r] => 'r"
10   assumes pluscomm: "[| a:R; b:R |] ==> plus a b = plus b a"
11   and plusassoc: "plus (plus a b) c = plus a (plus b c)"
12   fixes zero :: "'r"
13   assumes zeroinR: "zero : R"
14   and zeroisid_r: "a:R ==> plus a zero = a"
15   fixes negative :: "'r => 'r"
16   assumes negcancels: "a:R ==> plus a (negative a) = zero"
17   assumes timescomm: "[| a:R; b:R; c:R |] ==>
18     times a (times b c) =
19     times (times a b) c"
20   and distleft: "[| a:R; b:R; c:R |] ==>
21     times a (plus b c) =
22     plus (times a b) (times a c)"
23   and distright: "[| a:R; b:R; c:R |] ==>
24     times (plus b c) a =
25     plus (times b a) (times c a)"
26   assumes plusclosed: "[| a:R; b:R |] ==> plus a b : R"
27   and timesclosed: "[| a:R; b:R |] ==> times a b : R"
28 theorem (in ring)
29   assumes "a:R" and "b:R"
30   shows
31     "times a zero = times zero a \<and> times zero a = zero"
32     \<and>
33     times a (negative b) = times (negative a) b
34     \<and> times (negative a) b = negative (times a b)"
35   proof -
36     have "plus zero (times a zero) = times a zero"
37       sorry
38     also have "... = times a (plus zero zero)"
39       sorry
40     also have "... = plus (times a zero) (times a zero)"
41       sorry
42     also have "zero = times a zero"
43       sorry
44     have "times a zero = zero"
45       sorry
46   have "plus (times a (negative b)) (times a b)"
```

```
67           = times a (plus (negative b) b)"  
       sorry  
69   have "times a (plus (negative b) b) = times a zero"  
       sorry  
71   have "times a zero = zero"  
       sorry  
73  
75   have "times a (negative b) = negative (times a b)"  
       sorry  
  
77   show ?thesis sorry  
qed
```

Figure 2.13: Automatically generated Isabelle skeleton using CGa and DRa annotations.

2.2.6.5 Finishing of the halfbaked proof

The halfbaked proof still needs some extra bits of information to finish the final proof. This section removes the ‘sorry’ commands and completes the proofs through the document so that it is fully parsable by Isabelle. This final step is done entirely by hand and there is no automation to help the user finish this last step.

```
5  locale ring =
6    fixes R :: "'r set"
7    assumes nonempty: "R \<noteq> {}"
8    fixes plus :: "'r, 'r] => 'r"
9    fixes times :: "'r, 'r] => 'r"
10   assumes pluscomm: "[| a:R; b:R |] ==> plus a b = plus b a"
11   and plusassoc: "plus (plus a b) c = plus a (plus b c)"
12   fixes zero :: "'r"
13   assumes zeroinR: "zero : R"
14   and zeroisid_r: "a:R ==> plus a zero = a"
15   fixes negative :: "'r => 'r"
16   assumes negcancels: "a:R ==> plus a (negative a) = zero"
17   assumes timescomm: "[| a:R; b:R; c:R |] ==>
18     times a (times b c) =
19     times (times a b) c"
20   and distleft: "[| a:R; b:R; c:R |] ==>
21     times a (plus b c) =
22     plus (times a b) (times a c)"
23   and distright: "[| a:R; b:R; c:R |] ==>
24     times (plus b c) a =
25     plus (times b a) (times c a)"
26   assumes plusclosed: "[| a:R; b:R |] ==> plus a b : R"
27   and timesclosed: "[| a:R; b:R |] ==> times a b : R"
28 theorem (in ring)
29   assumes "a:R" and "b:R"
30   shows
31     "times a zero = times zero a \<and> times zero a = zero"
32     \<and>
33     times a (negative b) = times (negative a) b
34     \<and> times (negative a) b = negative (times a b)"
35   proof -
36     have "plus zero (times a zero) = times a zero"
37       sorry
38     also have "... = times a (plus zero zero)"
39       sorry
40     also have "... = plus (times a zero) (times a zero)"
41       sorry
42     also have "zero = times a zero"
43       sorry
44     have "times a zero = zero"
45       sorry
46   have "plus (times a (negative b)) (times a b)"
```

```
locale ring =
6   fixes          R :: "'r set"
8     assumes nonempty: "R \<noteq> {}"
8   fixes          plus :: "'r, 'r] => 'r"
8   fixes          times :: "'r, 'r] => 'r"
10  assumes pluscomm: "[| a:R; b:R |] ==> plus a b = plus b a"
10    and plusassoc: "plus (plus a b) c = plus a (plus b c)"
12  fixes          zero :: "'r"
12  assumes zeroinR: "zero : R"
14    and zeroisid_r: "a:R ==> plus a zero = a"
14  fixes          negative :: "'r => 'r"
16  assumes negcancels: "a:R ==> plus a (negative a) = zero"
16  assumes timescomm: "[| a:R; b:R; c:R |] ==>
18                times a (times b c) =
18                times (times a b) c"
20    and distleft: "[| a:R; b:R; c:R |] ==>
20                  times a (plus b c) =
22                  plus (times a b) (times a c)"
22    and distright: "[| a:R; b:R; c:R |] ==>
24                  times (plus b c) a =
24                  plus (times b a) (times c a)"
26  assumes plusclosed: "[| a:R; b:R |] ==> plus a b : R"
26    and timesclosed: "[| a:R; b:R |] ==> times a b : R"
28    and negclosed: "a:R ==> negative a : R"

theorem (in ring)
71 assumes "a:R" and "b:R"
71 shows
73   "times a zero = times zero a \<and> times zero a = zero
73     \<and>
75   times a (negative b) = times (negative a) b
75     \<and> times (negative a) b = negative (times a b)"
```

```
    by simp
96   also from zeroinR zeroisid_l [of zero]
      have "... = plus (times zero a) (negative (times zero a))" "
98     by simp
      also from prems zeroinR timesclosed [of zero a]
          negcancels [of "times zero a"]
      have "... = zero" by simp
102   finally show ?thesis .
qed
104   have A: "times a zero = times zero a"
proof -
106     have "plus zero (times a zero)
           = plus (times a zero) (times a zero)"
108   proof -
      from prems zeroinR timesclosed [of a zero]
      zeroisid_l [of "times a zero"]
      have "plus zero (times a zero) = times a zero" by auto
110   also from zeroinR zeroisid_l [of zero]
      have "... = times a (plus zero zero)" by auto
112   also from prems zeroinR distleft [of a zero zero]
      have "... = plus (times a zero) (times a zero)" by auto
114   finally show ?thesis .
116 qed
118   from prems zeroinR timesclosed [of a zero]
      this plus_cancels [of zero "times a zero" "times a zero"]
120   have "zero = times a zero" by auto
      from this B show "times a zero = times zero a" by auto
122 qed
124   have D: "times (negative a) b = negative (times a b)"
proof -
126     have "plus (times (negative a) b) (times a b) = zero"
128   proof -
      from prems negclosed [of a] distright [of b "negative a" a]
      have "plus (times (negative a) b) (times a b)
            = times (plus (negative a) a) b"
130     by auto
      also from prems negclosed [of a] pluscomm [of "negative a" a]
      negcancels [of a]
      have "... = times zero b" by auto
132   also have "... = zero"
proof - -- "This is slightly modified from the proof for B"
134   from prems zeroinR timesclosed [of zero b]
      zeroisid_r [of "times zero b"]
136   have "times zero b = plus (times zero b) zero" by auto
      also from prems this zeroinR timesclosed [of zero b]
138       negcancels [of "times zero b"]
      have "... = plus (times zero b)
           (plus (times zero b) (negative (times zero b)))"
140       by auto
142   also from this plusassoc [of "times zero b" "times zero b"]
```

```

                                "negative (times zero b)"]
146      have "... = plus (plus (times zero b) (times zero b))
           (negative (times zero b))" by auto
148      also from prems zeroinR distright [of b zero zero]
           have "... = plus (times (plus zero zero) b)
150                      (negative (times zero b))"
           by simp
152      also from zeroinR zeroisid_l [of zero]
           have "... = plus (times zero b) (negative (times zero b))"
154      by simp
           also from prems zeroinR timesclosed [of zero b]
156          negcancels [of "times zero b"]
           have "... = zero" by simp
158      finally show ?thesis .
           qed
160      finally show ?thesis .
           qed
162      from this
           have "plus (plus (times (negative a) b) (times a b))
164                      (negative (times a b))
           = plus zero (negative (times a b))"
166      by auto
           from this plusassoc
168      have "plus (times (negative a) b) (plus (times a b)
           (negative (times a b)))
170          = plus zero (negative (times a b))"
           by auto
172      from prems this timesclosed [of a b]
           negcancels [of "(times a b)"] negclosed [of a]
174      zeroisid_r [of "times (negative a) b"]
           timesclosed [of "negative a" b]
176      have "times (negative a) b = plus zero (negative (times a b))"
           by auto
178      from prems this timesclosed [of a b] negclosed [of "times a b"]
           zeroisid_l [of "negative (times a b)"]
180      have "times (negative a) b = negative (times a b)" by auto
           from prems this show ?thesis by auto
182      qed
           have C: "times a (negative b) = times (negative a) b"
184      proof -
           have "plus (times a (negative b)) (times a b) = zero"
186      proof -
           from prems negclosed [of b] distleft [of a "negative b" b]
           have "plus (times a (negative b)) (times a b)
           = times a (plus (negative b) b)"
188      by auto
           also from prems this negclosed [of b] negcancels [of b]
190          pluscomm [of "negative b" b]
           have "... = times a zero" by auto
192          also from this A B

```

```

        have "... = zero" by auto
196      finally show ?thesis by auto
qed
198      from this
        have "plus (plus (times a (negative b)) (times a b))
200          (negative (times a b))
              = plus zero (negative (times a b))" by auto
from prems this plusassoc timesclosed [of a b]
204      negcancels [of "times a b"]
        have "plus (times a (negative b)) zero
206          = plus zero (negative (times a b))" by auto
from this zeroisid_l [of "negative (times a b)"]
208      zeroisid_r [of "times a (negative b)"] prems
timesclosed [of a b] negclosed [of "times a b"]
negclosed [of b] timesclosed [of a "negative b"]
212      have "times a (negative b) = negative (times a b)" by auto
from this D
214      have "times a (negative b) = times (negative a) b" by auto
from this show ?thesis by auto
qed
216      from A B C D show ?thesis by auto
6218 qed

```

Figure 2.14: Manually finished Isabelle proof.

2.2.7 Conclusion

This section provides detailed information on the framework this thesis will be based on. We have described a brief history of where the original MathLang framework stemmed from and an overview of the methodology. We describe the CGa, DRa in detail and give a brief view of the TSa which has been used to formalise the common mathematical text which is what mathematical documents are usually written in. We have shown examples of the documents produced in each step whether they were produced automatically or manually. A full worked example was shown from the original text into a fully parseable and checked Isabelle proof.

This thesis follows similar steps from MathLang as MathLang is already a working proven concept which translates a mathematical text into a full proof. Since Z is a subset of Mathematics the steps should work for Z specifications as well.

We will use the same steps to check the correctness of a software specification albeit each step in the MathLang framework will be adapted, changed and imple-

mented to suit the software specification syntax.

2.3 Formal Methods and Languages

Formal methods are usually used to assist in formalising a variety of texts including systems, software and even language itself.

Definition 2.3.1 (Formal Language). *A language designed for use in situations in which natural language is unsuitable, as for example in mathematics, logic, or computer programming.⁵*

Definition 2.3.2 (Formal Specification). *The specification of a program's properties in a language defined by a mathematical logic.⁶*

Definition 2.3.3 (Formal methods). *Mathematical approaches to software and system development which support the rigorous specification, design and verification of computer systems.⁷*

Formal methods are mathematical approaches to software and system development which support the rigorous specification, design and verification of computer systems [?]. 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 rigour using formal methods eliminates design errors earlier and results in substantially reduced time [?].

⁵www.dictionary.com/browse/formal-language

⁶www.wiki.c2.com/?FormalSpecification

⁷www.fmeurope.org/?page_id=2

2.3.1 A brief history of formal methods

The first known formal language is thought to be used by Frege in his *Begriffschrift* (1879), *Begriffschrift* meaning ‘concept of writing’ described as ‘formal language of pure thought’. Frege formalised propositional logic as an axiomatic system.

Formal methods then grew in the following:

- 1940’s, Alan Turing annotated the properties of program states to simplify the logical analysis of sequential programs
- 1960’s Floyd, Hoare and Naur recommended using axiomatic techniques to prove programs meet their specification.
- 1970’s Dijkstra used formal calculus to aid development of non-deterministic programs

Formal methods have a large presence in academia and have also made their way into various industries to prevent design flaws in high integrity systems. Previous design errors have been found in systems such as the Therac-25 machine, which was used for radiation therapy produced by Atomic Energy of Canada Limited in 1982. It was involved in multiple incidents in which patients were given massive overdoses of radiation [?]. Another major fault which led to disastrous results was NASA’s Checkout Launch and Control System (CLCS) cancelled 9/2002 ⁸.

2.3.2 Types of formal methods

Today there are many different types of formal methods used both in industry and academia. Specification languages are expressive languages with general proof methods, such as VDM, Z, B. Another type of formal method could be program correctness proofs which associates logical inference rules with programming syntax, e.g. Hoare triples and Gries Methodology. There may also be model based approaches to formal methods, which are domain specific languages with precise algorithms for correctness proofs, e.g. Temporal logic [?], Fuzzy logic.

⁸www.spaceref.com/news/viewnews.html?id=475

Formal methods are used to precisely communicate specifications and the function of programs. They are also used to ensure the correctness of systems particularly safety critical systems.

Formal methods have been a success in a variety of projects. For example, in the Sholis project [?], 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 [?] 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.

Another case study where formal methods have been used in industry is the NASAs Mars Science Laboratory Mission (MSL) [?]. This system relied on various different mechanisms to command the spacecraft from earth and to understand the behavior of the rover and spacecraft itself. The paper suggested that '*test engineers cannot eyeball the hundreds of thousands of events generated in even short tests of such a complex system*'. Therefore runtime verification using formal specification offered a solution to this problem.

Other projects which formal methods are commonly used are to explain policy management systems for Unmanned Air Vehicles [?] and Connected and Autonomous Vehicles [?]. In these papers, the formal methods were used as a bridge between the system engineers designing the requirements and the developers who were developing the software. As both these systems were classed as high integrity, the formal methods were used to reduce ambiguity and define precisely what the system was required to do.

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 challenge (published in 2014) as the ‘adoption hurdle’ is perceived too high. Customers and regulators have taken a variety of stances on static analysis 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 analysis could be simplified and broken up into smaller more understandable steps and thus would allow more users to take on the idea.

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 [?].

So far we have seen what are the different types of formal methods, where formal methods have been used, and why some people are still reluctant to use them. So what needs to be done to make formal methods industrial strength? Nirmal Pandey [?] suggests the following:

- 1 Bridge gap between real world and mathematics
- 2 Mapping from formal specifications to code (preferably automated)
- 3 Patterns identified
- 4 Level of abstraction should be supported
- 5 Tools needed to hide complexity of formalism
- 6 Provide visualization of specifications
- 7 Certain activities not yet formulisable methods
- 8 No one model has been identified which should be used for software)

Table 2.2: Nirmal Pandey's suggestions of how to make formal methods industrial strength.

2.3.3 Conclusion

In this section we have identified the differences between formal methods, formal languages and formal specification. We have seen how formal methods originated from mathematics and how it grew over time to become what it is today. We see a connection with Frege's work on mathematical notation and identified it as the first formal notation found in history. We identified there are a variety of formal methods and new methods are even being developed today to comply with the systems in questions. For example MSL had it's own specification language developed for verification. Despite all the advantages, some managers are still reluctant to use formal methods in their system development and thus as Pandey suggested there still needs to be some work done on making formal methods industrial strength. One of his suggestions was that tools are needed to hide complexity of formalisms and to provide visualization of specifications, which this thesis addresses.

2.4 Z Syntax and semantics

In this section we give a brief overview on the core fundamentals of the Z formal language. We outline a basis of propositional and predicate logic and describe how Z syntax is formed to specify a system.

2.4.1 Introduction to Z

Z is based on predicate Calculus, Zermelo-Frankel set theory as we introduced at the beginning of this chapter in section ??.

It is a particular formal method which was developed to specify the new Customer Information Control System (CICS) functionality [?]. 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 [?].

2.4.2 Propositional and predicate logic

Z specifications are built using predicate and propositional logic.

2.4.2.1 Propositional logic

Propositional logic works with statements which must be either true or false but can not be both. The following are propositional statements:

- A tree is green
- A tree has leaves
- All plants have flowers

Propositions can be connected in a variety of ways. Figure ?? shows a table of logical connectors in order of operator precedence.

\neg	negation	not
\wedge	conjunction	and
\vee	disjunction	or
\Rightarrow	implication	implies
\Leftrightarrow	equivalence	if and only if

Table 2.3: Logical connectors giving the symbol, its name and pronunciation.

We can now build compound propositions, which are propositional statements joined together using these connectors, e.g.

- the glass is full \wedge the glass is clear
- the phone isn't working \vee the phone battery has died
- the sun is shining \Rightarrow I don't need a raincoat

2.4.2.2 Predicate logic

Predicate logic allow us to make statements which describe properties that must be satisfied by every, some or no objects in some universe of discourse. Examples are:

- Every plane can fall out of the sky.
- At least one cloud has a silver lining.
- Jake knows all rugby players in the Scottish team.

To formalise such statements we require a language of predicate calculus. A predicate is a statement with a place for an object. A predicate can turn into a proposition once we put an object in it, therefore we can not say whether it is true or false once the predicate has filled in missing information. For example we can say ‘ $x > 5$ ’ is a predicate but not a proposition until we know what ‘ x ’ is. We can make a proposition out of ‘ $x > 5$ ’ by putting a *quantifier* with it. So we can say, ‘there is an x which is larger than 5’. This is written formally as ‘ $\exists x > 5$ ’. This statement can now be written in our Z syntax.

We can now formalise one of our previous predicate statements:

$\exists c : CLOUD \bullet c \text{ has silver lining}$ This statement says that there exists c which is a cloud and c has a silver lining.

2.4.3 Sets and Types

A *set* is a collection of distinct objects called *elements*. For example the set of all people. Every expression in a Z specification belongs to a set called its *type* and whenever we introduce a variable we must declare its type [?]. For example:

[HUMAN] the set of all humans

\mathbb{N} the set of all natural numbers

Another way of introducing types is using a *free type definition*, where one enumerates the names of the elements in that type, for example:

$SHAPES ::= square \mid circle \mid triangle$

$PLANTS ::= tree \mid shrub \mid herb$

In Z specifications, we introduce variables before they are used in the expressions

(predicates) by a means of writing a *declaration*. A declaration can introduce either one or multiple variables. For example:

ralph : *HUMAN*

a, b, c : \mathbb{N}

2.4.4 Structure of a Z specification

In the Z notation there are two languages: the mathematical 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 ??) or vertically (figure ??).

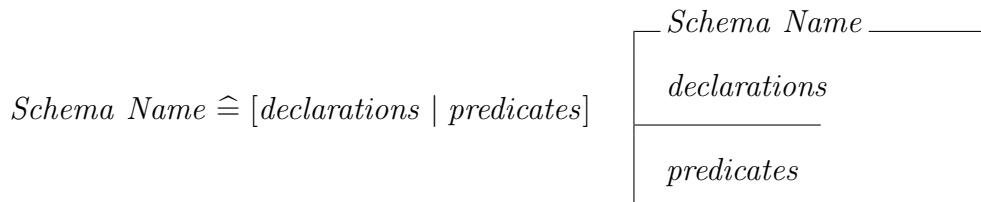


Figure 2.15: An example of a schema written horizontally.

Figure 2.16: 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:



We can also introduce global variables by means of an *axiomatic description* in Z. These global variables may be referred to throughout the specification. An example this Z construct is as follows:

$$\begin{array}{c} k : \mathbb{N} \\ \hline k = 100 \end{array}$$

The axiomatic description in the Z specification means that whenever the global variable k is referred to, it will always represent the natural number 100.

The full language of Z can be explored in [?], [?] and [?].

2.4.5 A full example in Z

Here is a small schema which tells a story about a child called Ralph and his mum. We declare the types $OBJECT$ which is a type containing all objects. We also declare the type $EMOTION$ which consists of 3 emotions: *happy*, *sad* and *angry*

$[OBJECT]$

$EMOTION ::= happy \mid sad \mid angry$

In this first schema we declare the state variables which is ralph and mum who are of type human, colourIn which maps a human to an object (representing that particular human is colouring in that object) and feeling which maps a human to an emotion (representing that human is feeling that emotion).

In the state we say that ralph is always feeling happy

$ColouringIn$

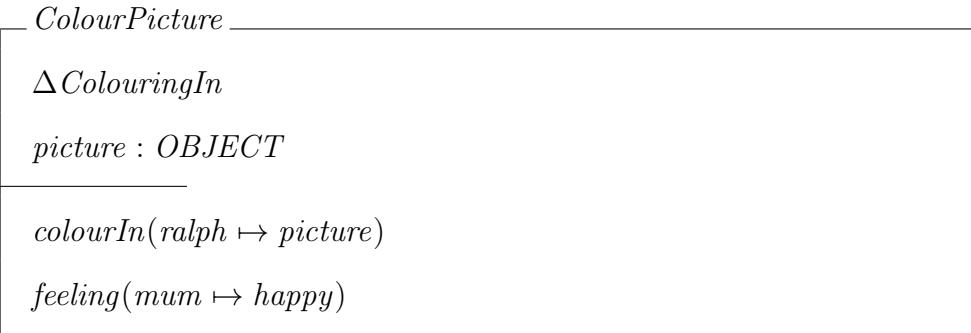
$ralph, mum : HUMAN$

$colourIn : (HUMAN \rightarrow OBJECT)$

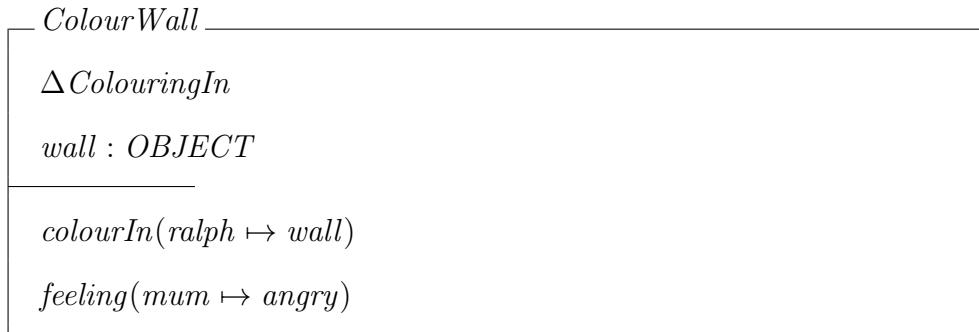
$feeling : (HUMAN \rightarrow EMOTION)$

$feeling(ralph \mapsto happy)$

In this schema we say that when ralph is colouring in a picture, mum is feeling happy. We use a Δ to signify this schema changes the current state.



However in this schema, we say we say when ralph is colouring the wall, then mum is angry.



2.4.6 Conclusion

In this section we gave a brief introduction to Z. We described propositional and predicate logic as well as defining the structure of a Z specification. We gave a full example of a Z specification and explained each part and what it means. In the next section we identify various proving systems which currently exist for mathematics, other formal methods and Z. We have chosen to translate Z specifications as they are the root of many other formal notations (object Z, B-Method etc.) Z specifications can also accommodate informal text being written around the formal notation.

2.5 Proving systems

Before the day's of mechanical computers the only way to check a mathematical documents logic was correct was using hand written proofs. Then one can ask another mathematician such as a peer or an expert in the field to check over ones mathematics to prove it was correct. However even with more than one mathematician checking over the mathematical logic, one could not be 100% certain it was correct due to human error.

Once mechanical computers were developed, work started on proof assistants (or proof checkers) and automated theorem provers. Checking the correctness using automated theorem provers adds an extra level of certainty that the mathematics written is correct, thus it is a good idea to check the correctness of systems which are high integrity.

2.5.1 Levels of Rigour

Depending on the integrity of the system being developed there are various formal techniques which could be used to mathematically verify the system design and implementation satisfy the functional and safety properties. These specifications and verifications may be done using various degrees of rigour such as:

- Level 1 represents the use of mathematical logic to specify the system.
- Level 2 uses pencil-and-paper proofs.
- Level 3 consists of formal proofs being checked by a mechanical theorem prover.

Level 1 can be done using various levels of abstraction to describe the system. One abstract level would be to describe the abstract properties of the system and another abstract level could be to describe the implementation in an algorithmic way.

Level 2 goes beyond level 1 in that the pencil-and-paper proofs are used to describe that the lower level properties imply the abstract properties in level 1.

Level 3 is the highest level of rigour one can do to check the proofs of a specification. It consists of using an autonomic theorem prover to confirm that all proofs

are valid. The level 3 process of convincing a mechanical prover is really a process of developing an argument for an ultimate sceptic who must be shown every detail [?].

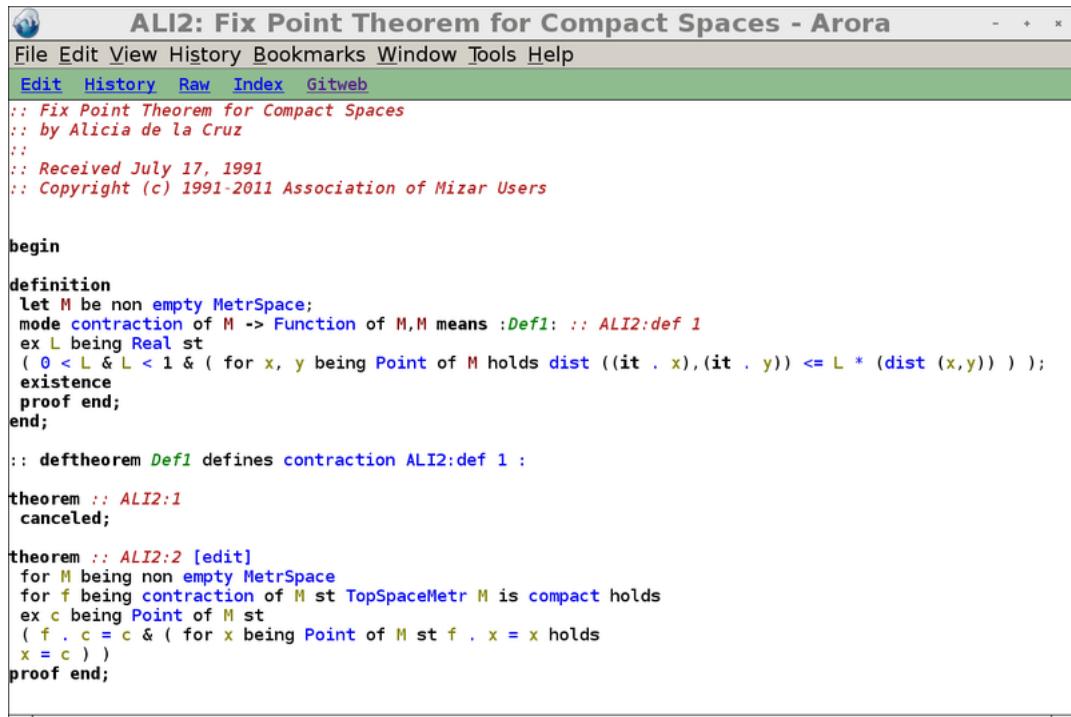
2.5.2 Proving systems for maths

In section ?? we saw how mathematics has evolved and become the complex system it is today. Since the rise of computers the ability to check some of these old and new theories automatically has been possible.

Mizar [?], Isabelle [?] and Coq [?] are types of **Proof assistants** (or proof Checkers). Whilst PVS and Vampire are types of **theorem provers**. Both proof assistants and theorem provers can be classed as ‘proof systems’. Each proof system provides its own formal language for writing mathematics based on some foundation of logic.

The first project which used a ‘computer’ for automated verification dates back to 1967. N.G. de Bruijn started the AutoMath Project (described in section ??). A few years later Trybulec created his project Mizar [?], where he aimed to develop a computerised assistant to edit mathematical papers.

From 1989 the works of the Mizar project worked on improving the Mizar system and developing the database for mathematics. The Mizar Language is a formal language which is derived from the mathematical vernacular. The author wanted to design a language readable for mathematics and rigorous enough to enable processing and verifying by computer software.



```

ALI2: Fix Point Theorem for Compact Spaces - Arora
File Edit View History Bookmarks Window Tools Help
Edit History Raw Index Gitweb
:: Fix Point Theorem for Compact Spaces
:: by Alicia de la Cruz
::
:: Received July 17, 1991
:: Copyright (c) 1991-2011 Association of Mizar Users

begin
definition
let M be non empty MetrSpace;
mode contraction of M -> Function of M,M means :Def1: :: ALI2:def 1
ex L being Real st
( 0 < L & L < 1 & ( for x, y being Point of M holds dist ((it . x),(it . y)) <= L * (dist (x,y)) ) );
existence
proof end;
end;

:: deftheorem Def1 defines contraction ALI2:def 1 :

theorem :: ALI2:1
canceled;

theorem :: ALI2:2 [edit]
for M being non empty MetrSpace
for f being contraction of M st TopSpaceMetr M is compact holds
ex c being Point of M st
( f . c = c & ( for x being Point of M st f . x = x holds
x = c ) )
proof end;

```

Figure 2.17: Screenshot of the Mizar theorem prover [?].

Isabelle [?] is a generic proof assistant. It accepts a variety of mathematics into it's syntax and also has it's own formal language and provides tools for those formulas in a logical calculus. The main application is the formalization of mathematical proofs and formal verification which includes proving the correctness for computer hardware or software.

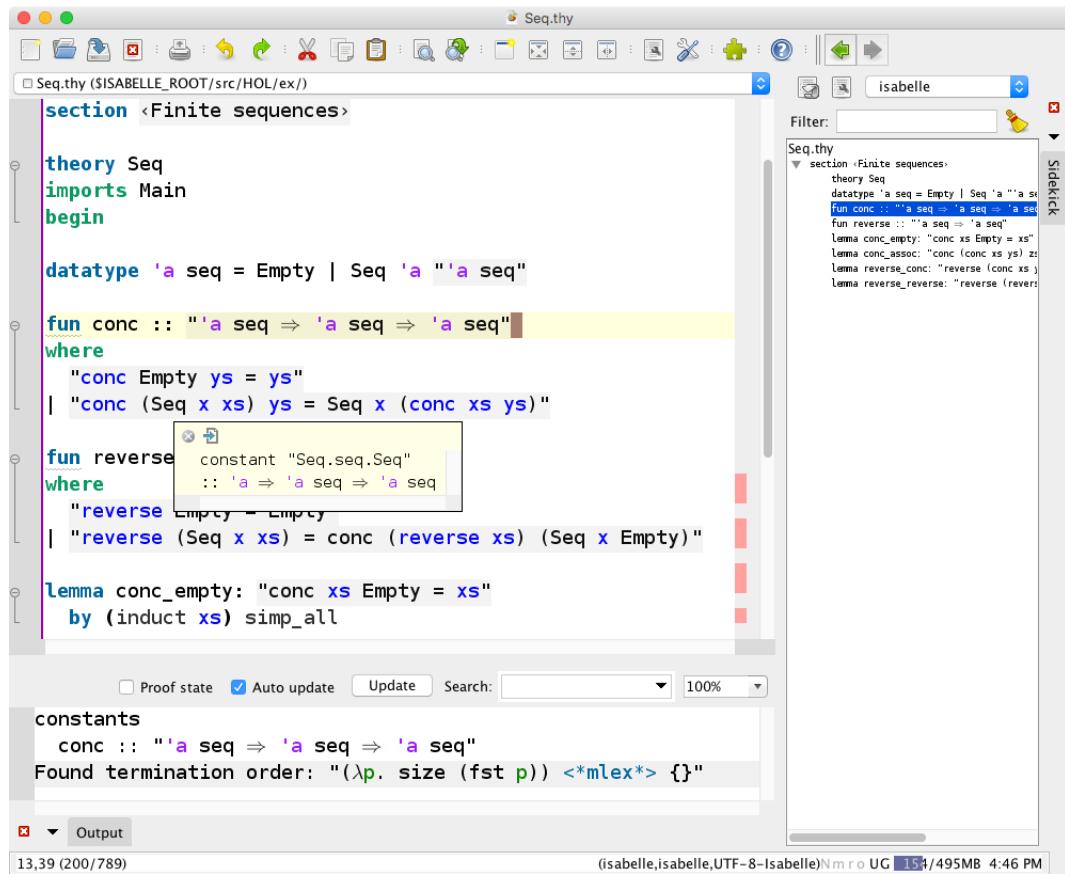


Figure 2.18: Screenshot of the Isabelle theorem prover [?].

A third proof assistant is Coq [?], which also provides its own formal language to write mathematical definitions, executable algorithms and theorems for semi-interactive development of machine checked proofs. Although it names itself a 'formal proof management system', like Isabelle and Mizar it is used to verify and check properties of software and hardware systems and the formalization of mathematics.

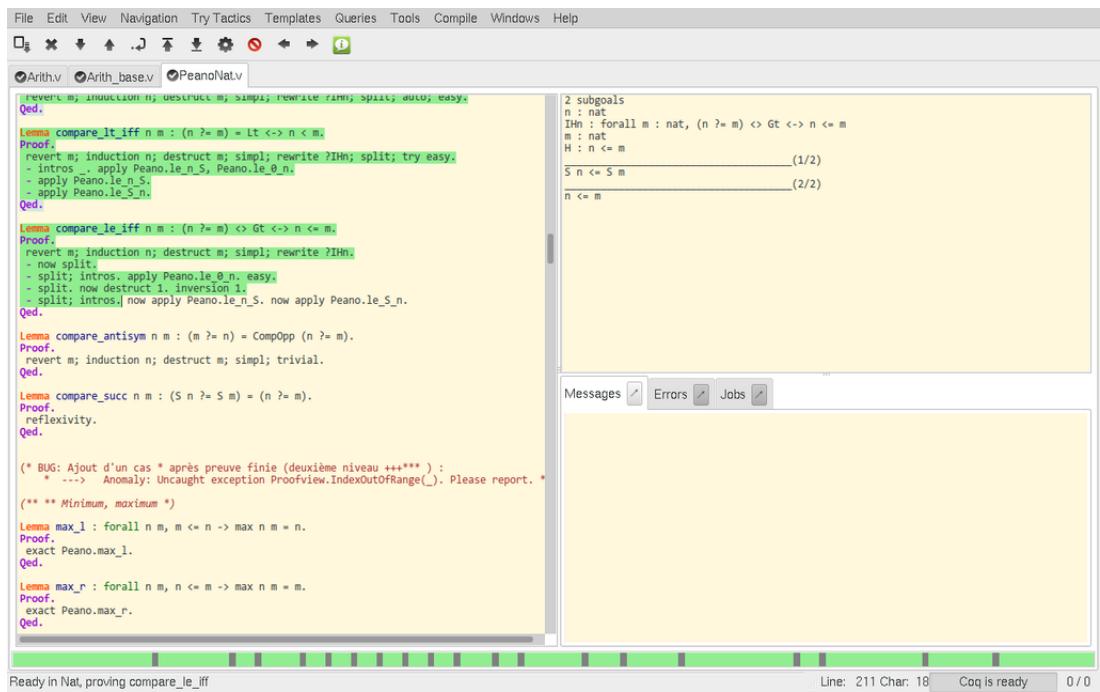


Figure 2.19: Screenshot of the Coq theorem prover [?].

There are many other proof assistants for general mathematics (Nuprl, Otter etc.) and generally they all support checking for correctness of mathematical theories. However, a disadvantage of using these systems is the enormous expense of formalisation in any proof system.

A mathematician who wants to verify the logic of a document has to decide on a proof system to use. All proof systems have their own advantages and disadvantages. The mathematician then needs to learn and create a document with the syntax of his chosen proof system.

Also, not all proof system have meaningful support for the common mathematical language or a generic formal specification language. There may be proof systems built for specific formal specification language but not one which supports all. The rigid structure and rules for formal documents created in proof systems are more closer to a computer programming language then to natural language. An exception of this is Mizar, however even natural language written in Mizar has to be structural and rigid.

In summary, computerising any form of mathematical document using a proof system is similar to writing a computer program. Where the formal language of

the proof system corresponds a programming language. Therefore, mathematicians require to have some form of computer programming knowledge. A proof system can check the logical correctness of a document if it is detailed enough for the software to parse, therefore the computerisation of the mathematical document becomes labour intensive and tedious for the author to write.

We have chosen to translate our specification into Isabelle as Isabelle has a lot of support and information online, and the syntax of Isabelle is very similar to the notation of Z.

2.5.3 Proving systems for specific formal method

Dafny [?] 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 rigour 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 Abrial [?] may be preferred in this case.

Another attempt at checking for correctness was written by Rex L.Page in Engineering Software Correctness [?]. 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 emphasizes 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 necessary and no new programming language is needed.

PVS (Prototype Verification System) [?] is an environment for constructing specifications and developing proofs which have been mechanically verified. PVS has its 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.

There are many other tools for Z which can be found on the Z Notation Wikia page [?]. For this thesis we will concentrate on translating Z specifications into theorem prover Isabelle [?]. 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 tool-kit (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 ([?]) therefore we can use parts of that research to aid the research in this thesis.

2.5.4 Properties to prove

Definition 2.5.1. *Proof Obligation: 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 [?].*

Therefore a proof obligation is a logical formula 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 consistent.

Woodcock and Cavalcanti [?] use the alphabetized relational calculus to give notational semantics to different constructs from programming patterns. This paper describes ‘*healthiness conditions*’ which identifies properties that characterize

theories. Each one of these healthiness conditions represents a fact about the computational model for the programs being studied.

Example 2.5.1. *The variable ‘clock’ is an observation of the current time which is always moving onwards. The following predicate ‘B’ specifies the clock variable:*

$$B \hat{=} \text{clock} \leq \text{clock}'$$

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 as a Z specification itself. 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 projects written in Z in her paper a tale of two proofs [?]. She explains how the proof process is deeply 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 consistent 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 2.20: Different points to prove in a specification [?].

Some of the points in figure ?? can be automated such as ‘consistency checks’ and ‘sanity 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 and 2 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 [?] 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 consistency lemmas to prove. The user can then choose to prove these lemmas and the proof audit their theorem. However, even if proof auditing 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 [?], some clients wouldn’t need that amount of rigour 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 [?].

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

Example 2.5.2.

ResourceManager

free : FN

Any resource that is free can be allocated

Allocate

$\Delta ResourceManager$

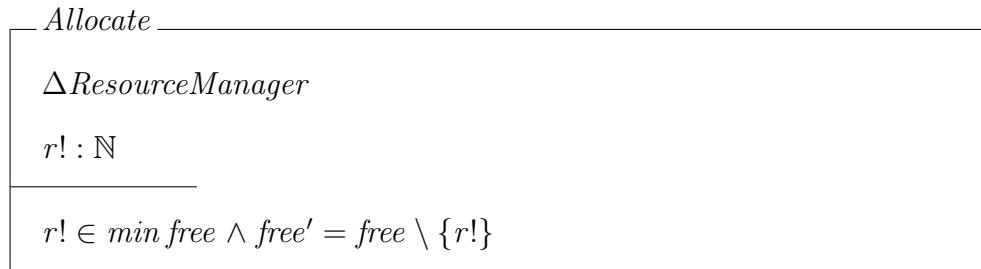
$r! : \mathbb{N}$

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

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

Example ?? 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 2.5.3.



Refining an abstract specification which is described in [?] and in [?] is exactly what Stepney points out in point 5 from figure ???. To show that this property holds the user would need to produce another specification which refines their original one and then include properties of how they relate to each other. 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 and 2 in figure ???. Items 3, 4 and 5 would be to difficult to automate and therefore out of the scope of this thesis.

Fraser and Banach [?] 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 interact easily with other tools in other kits. A more dynamic approach such as RODIN [?] and Overture [?] are perusing a more flexible approach. ZMathLang will need to follow in the footsteps of these two tool-kits in order to be successful and not commit to a particular system until needed. This is why the steps 1-3 of the translation path (figure ??) 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 tool-kit was to support retrenchment, which is a formal technique used along side refinement. The new tool-kit was created which was able to use a variety of techniques in a single working environment. The frog tool-kit can prove the relationships between multiple specifications (which we see as point 5 in figure ?? from Stepney [?]). 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 tool-kit.

Since formal specifications are not executable it is difficult to verify the consistency of the specification. Wen, Miao and Zeng [?] present and approach to generate proof obligations to check for consistency of object Z specifications. Checking for consistency of specifications is described in point 2 in figure ???. 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 initialized the beginning state of the specification.

An operation can transform 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 [?] 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.

2.5.5 Conclusion

In this section we have highlighted the levels of rigour which describe systems and/or software used in industry. We also describe proving systems which have been used in the past to check over general mathematics. Other proving methods and assistants have been developed for specific languages such as Dafny and PVS which we give examples of. There have also been proving systems and tools implemented which are specific for Z which we have described in this section. We have also added what are the important properties we wish to find out from system specifications and how they can be checked for correctness.

2.6 Background Conclusion

This chapter starts from the very beginning of mathematics. The reason behind the research presented in this thesis has its roots in the origins of mathematics and why it is important to so many people to verify and be confident in their own expertise whether it's mathematics, logic, science or systems. There has been many ways in which people have tried to prove the correctness of this work but it usually is labour intensive, hard work, and requires serious expertise not only in the subject of the work but also in the proving systems doing the checking. One of the ways to simplify the checking of mathematical correctness was the MathLang framework. This method has been described in this chapter and an entire example has been shown of how one could check the correctness of their mathematical texts. Even [?] had stated that future work of MathLang is that the framework should

be developed to cope with more mathematics, therefore adapting the framework for formal specifications would address this point.

So far MathLang checks for mathematical correctness, however there are many other texts and documents which checking for correctness would play a big part. The rapid growth in technology⁹ has meant that checking the specifications for correctness is important now more than ever. We described other tools and techniques which have been implemented to address this in section ???. We described where formal methods began and the different types of formal methods being used. By adapting an existing framework (MathLang) and changing some design and implementations so that it can be used for formal system/software specifications we can cover items 1, 3, 5, 6, and 7 from table ???.

Section ?? gave an overview of the Z formal specification language and describes what it is made from and how one can write a software specification in Z. We use the Z notation to base this thesis on as a lot of other formal notations (B-method, event B etc.) use Z as the basis for their syntax.

We have also described proving systems which have been used for mathematics, various formal methods and Z syntax. Each of these proving systems have their own advantages and disadvantages and it is always up to the user to choose the direction and prover they wish to use. Section ?? described important properties users may wish to check about their system specification. Not all users wish to check the same thing, and since systems all have various degrees of criticality they may have their own independent properties which may need to be checked in order to verify safety. Therefore this thesis will concentrate on sanity checks (item 2 in figure ??) as these would be suitable for all systems.

2.6.1 MathLang for Z

To address some of the points made by Stringfellow in [?] of how to make formal methods industrial strength, this thesis takes work made by Kamareddine et al [?] and work on Z notation [?] and brings these two works together to creates a novel

⁹<http://bigthink.com/think-tank/big-idea-technology-grows-exponentially>

way of checking the correctness of formal system specifications.

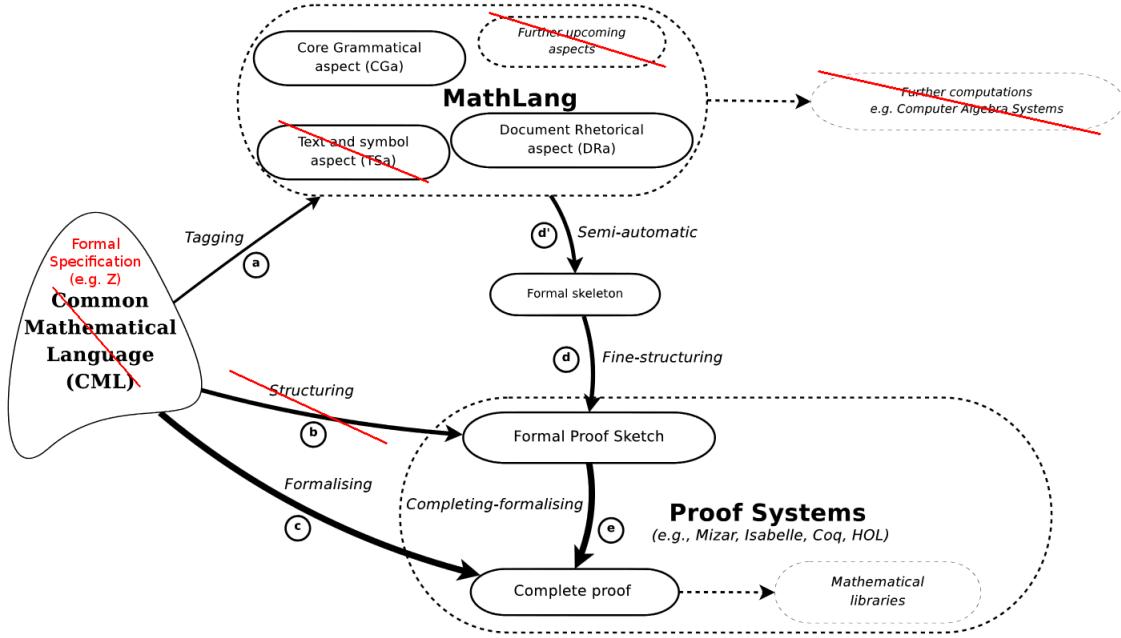


Figure 2.21: Adapting the original MathLang to accommodate formal system specifications.

By adapting the original MathLang framework to accommodate Z specifications, each aspect in the original framework will need to be redesigned and re-implemented. Figure ?? shows the original design for MathLang, however we have replaced ‘*common mathematical language*’ with ‘*formal specification*’. We removed the TSa as formal specifications are already written in a formal manner and may not need the TSa. We have also removed the arrow ‘*b*’ as there hasn’t been any work as yet which formal specifications are translated first to a skeleton and then a full proof.

Formal specifications have previously been fully verified with theorem provers by users which have knowledge on the formal specification language and the targeted theorem prover. By adapting the MathLang framework it should be possible to check the properties of formal specifications by users who have little or no expertise in the target theorem prover. Although the step-by-step methodology of translating texts into theorem provers is not new, the idea to apply it to formal specifications and the design and implementations to accommodate Z is and this is where we begin our thesis.

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 specification.

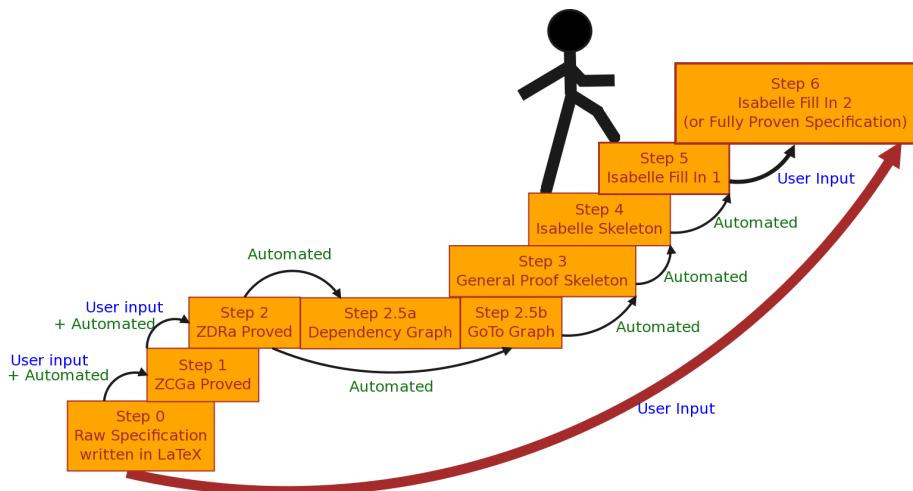


Figure 3.1: The steps required to obtain a full proof from a raw specification.

The framework is targeted at beginners in theorem proving. Users should have some idea of formal specifications but have no or little knowledge of the targeted theorem prover. Figure ?? 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 interchangeable 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 be done in the future.

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

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

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

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

This section gives an overview of each individual step in the ZMathLang tool-set.

3.1.1 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 its way to becoming formalised in Z). The specification should be written in L^AT_EX format and can be a mix of natural language and Z syntax. An example of a specification written in the Z notation can be seen in figure ??.

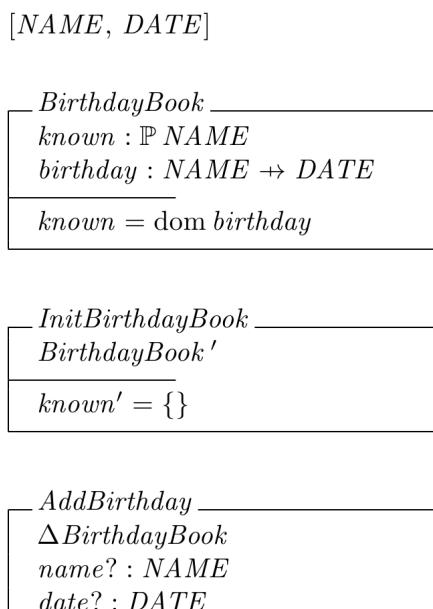


Figure 3.2: Example of a partial Z specification.

3.1.2 Step 1- The Core Grammatical aspect for Z

The next step in figure ?? shows that the specification should be ZCGa proved. Although this step is interchangeable with step 2 (ZDRA) it is shown as step 2 on the diagram for convenience. In this step the user annotates their document

which they have obtained in step 0 with 7 grammatical categories and then checks these for correctness. Figure ?? shows this step is achieved by user input and automation. The user input of this step is the annotations and the automation is the ZCGa checker. This automatically produces a document labeled with the various categories in different colours and can help identify grammar types to other members in the systems project team. A ZCGa annotated specification is shown in figure ???. The ZCGa is further explained in chapter ??.

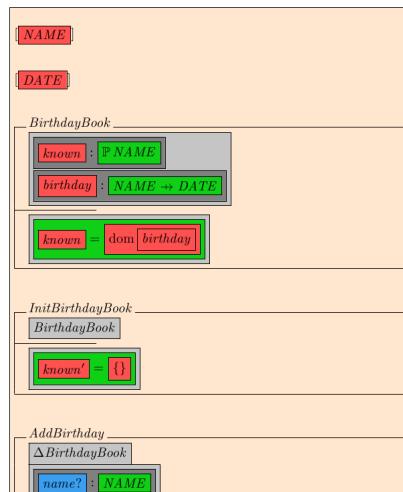


Figure 3.3: Example of a ZCGa annotated specification.

3.1.3 Step 2- The document Rhetorical aspect for Z

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

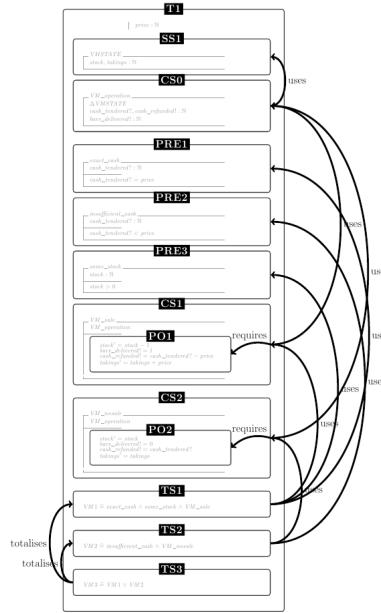


Figure 3.4: Example of a ZDRa annotated specification.

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

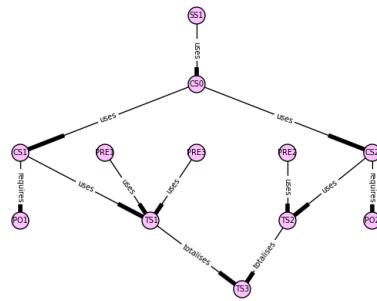
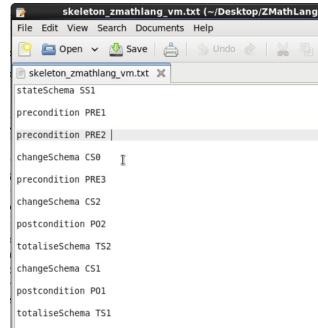


Figure 3.5: Example of an automatically generated goto graph.

3.1.4 Step 3- The General Proof skeleton

The following step is an automatically generated General Proof Skeleton aspect (Gpsa). This document is automated using the goto graph which is generated from the ZDRa annotated L^AT_EX 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 consistency of the specification i.e. the specification is consistent throughout. An example of a general proof skeleton is shown in figure ???. The Gpsa is further described in section ??.



```
skeleton_zmathlang_vm.txt (~Desktop/ZMathLang)
File Edit View Search Documents Help
skeleton_zmathlang_vm.txt
stateSchema S51
precondition PRE1
precondition PRE2 |
changeSchema CS0
precondition PRE3
changeSchema CS2
postcondition P02
totaliseSchema TS2
changeSchema CS1
postcondition P01
totaliseSchema TS1
```

Figure 3.6: Example of a general proof skeleton.

3.1.5 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, theory's 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 ???. Details of how this translation is conducted is described in chapter ?? of this thesis.

```
theory gpsazmathlang_birthdaybook
imports
Main
begin

record S51 =
(*DECLARATIONS*)

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

definition IS1 :: 
"(*IS1_TYPES*) => bool"
where
"IS1 (*IS1_VARIABLES*) == (P02)"

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 3.7: Example of an Isabelle skeleton.

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

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

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

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

3.1.7 Step 6- A fully proven Z specification

The final step in the ZMathLang framework (top of the stairs from figure ??), is to fill in the Isabelle file from step 5. This final step is represented by a slightly thicker arrow in figure ?? compared with the others as the user may need to have some theorem prover knowledge to prove properties. Also if there is some missing

information such as missing expressions and definitions the user must fill these out as well in order to have a fully proven specification. However this may be slightly easier then writing the specification from scratch as the user would already have examples of other instances in their Isabelle syntax form. More details on this last step is described in section ?? of this thesis.

3.2 Procedures and products within ZMathLang

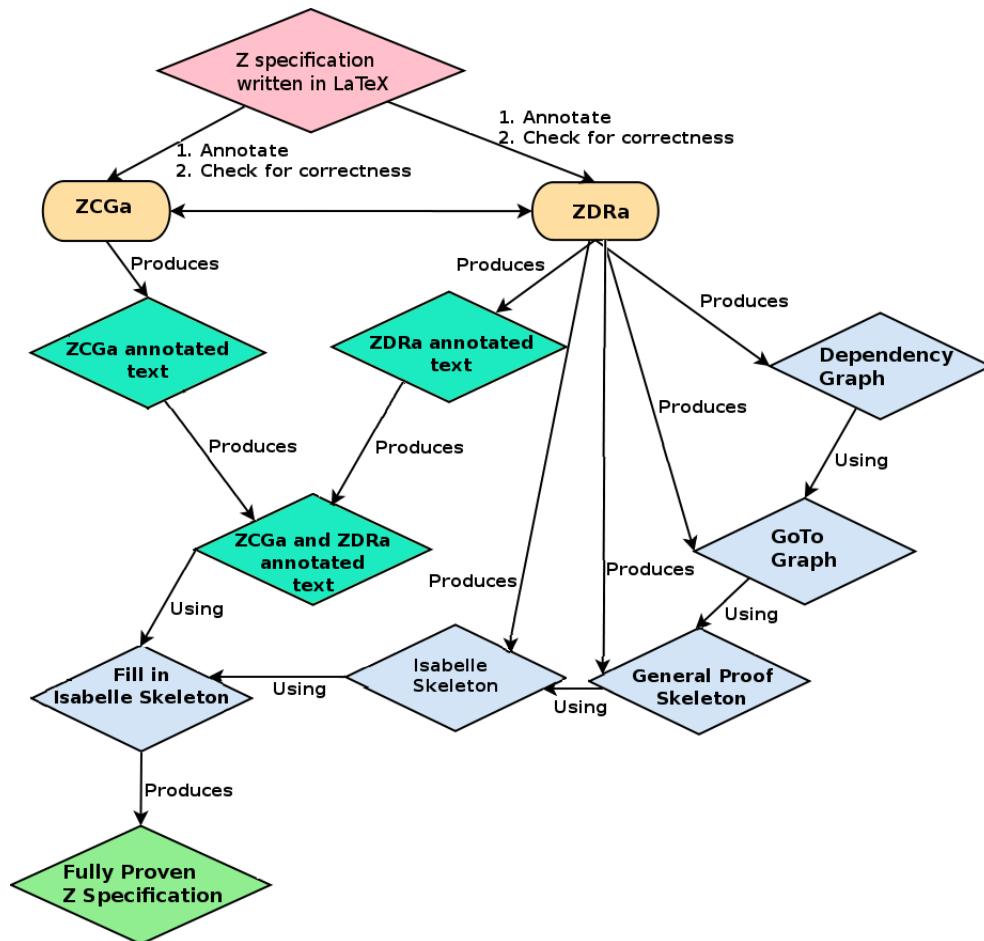


Figure 3.9: Flow chart of ZMathLang.

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

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

3.3 The ZMathLang LaTeX Package

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

3.3.1 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 3.10: Part of the syntax to define the colours for ZCGa in the ZMathLang L^AT_EX file.

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

3.3.2 L^AT_EX commands to identify ZDRa Instances

The ZDRa section of the L^AT_EX style file provides three new commands: `\draschema`, `\draline` and `\drathey`. The `\drathey` annotation is for the entire specification which contains all the instances 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.

<code>\draline{X}{\draschema{Y}{someContext}}</code>	Incorrect
<code>\draschema{Y}{\draline{X}{someContext}}</code>	Correct

The `\draline` annotation is to annotate any instance that is a line of text which contains plain text or ZCGa annotated text. But does not include any ZDRa annotated text. For example in figure ?? 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 ?? where the `\draline{PRE1}` annotation is embedded in the `\draschema{CS1}{` annotation.

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

Figure 3.11: Incorrect annotating of ZDRA.

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

Figure 3.12: Correct annotating of ZDRA.

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

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

Figure 3.13: The syntax to define a ZDRA schema instance in the ZMathLang L^AT_EX file.

The new command we are defining for `\draschema` is shown in figure ???. 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.

3.3.3 L^AT_EX commands to identify ZDRa Relations

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

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

Figure 3.14: The syntax to define a ZDRa schema relation in the ZMathLang L^AT_EX file.

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

3.3.4 L^AT_EX commands to identify ZCGa grammatical types

The ZCGa part of the L^AT_EX 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 3.15: The syntax to define a ZCGa grammatical categories.

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

3.4 Conclusion

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

Chapter 4

Z Core Grammatical aspect

The ZCGa is a weak type checker, which checks for grammatical correctness in formal Z specifications and partially formalised Z specifications. 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 its roots in weak type theory for mathematics [?]. Core grammatical correctness for Z has also adapted the rules from the CGa for mathematics (see section ?? in chapter ??).

This chapter focuses on the first step of the ZMathLang tool-set. 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 its own to check their specification for some level of rigor.

The first part of this chapter explains the design of the ZCGa and how the rules and categories have been adapted from [?]. 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 highlights some properties we can show about the ZCGa. Then we explain how the categories of the ZCGa syntax are converted into weak types and check Z specifications for grammatical 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 figure ??.

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	<i>variables</i>	$V = V^{\mathcal{T}} V^{\mathcal{S}}$	x
	<i>constants</i>	$C = C^{\mathcal{T}} C^{\mathcal{S}} C^{\mathcal{E}}$	c
	<i>binders</i>	$B = B^{\mathcal{S}} B^{\mathcal{E}}$	b
phrase	<i>terms</i>	$\mathcal{T} = C^{\mathcal{T}}(\vec{\mathcal{P}}) V^{\mathcal{T}}$	t
	<i>sets</i>	$\mathcal{S} = C^{\mathcal{S}}(\vec{\mathcal{P}}) B_{\mathcal{Z}}^{\mathcal{S}}(\mathcal{E}) V^{\mathcal{S}}$	s
sentence	<i>expressions</i>	$\mathcal{E} = C^{\mathcal{E}}(\vec{\mathcal{P}}) B_{\mathcal{Z}}^{\mathcal{E}}(\mathcal{E})$	E
	<i>definitions</i>	$\mathcal{D} = C^{\mathcal{S}}(\vec{V}) := \mathcal{S}$	D
discourse	<i>schematext</i>	$\Gamma = \emptyset \Gamma, \mathcal{Z} \Gamma, \mathcal{E}$	Γ
	<i>paragraphs</i>	$\Theta = \Gamma \triangleright \mathcal{E} \Gamma \triangleright \mathcal{D}$	θ
	<i>specifications</i>	$\text{Spec} = \emptyset \text{Spec}, \Theta$	spec
Other Category	abstract syntax	Meta-symbol	
<i>parameters</i>	$\mathcal{P} = \mathcal{T} \mathcal{S} \mathcal{E}$	P	
<i>declarations</i>	$\mathcal{Z} = V^{\mathcal{S}}:\text{SET} V^{\mathcal{T}}:\mathcal{S}$	Z	

Note: $\vec{\mathcal{P}}$ is a list of 0 or more \mathcal{P} 's, $\vec{\mathcal{S}}$ is a list of 0 or more \mathcal{S} 's,

$\vec{\mathcal{E}}$ is a list of 1 or more \mathcal{E} 's, \vec{V} is a list of 0 or more V 's.

Table 4.1: Categories of ZCGa syntax.

We use some of the weak types in [?] in our weak types for Z. 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 ??.

M	$m : \mathbb{N}$
	$n : \mathbb{P} \mathbb{N}$
	$m \geq 0$
	$\#n \geq 1$
M'	
M	
$m' : \mathbb{N}$	
$n' : \mathbb{P} \mathbb{N}$	
	$m' = 0$
	$n' = \{\}$

Figure 4.1: Basic example of a specification

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

variables The set of variables V is divided into two subsets V^T and V^S which correspond to variables giving terms and variables giving sets respectively.

- V^T . An example of a variable giving a term would be ‘ m ’ in figure ??.
- V^S . An example of a variable giving a set would be ‘ n ’ in figure ??.

constants The set of constants range over constants giving terms C^T , constants giving sets C^S and constants giving expressions C^E .

- C^T . An example of a constant giving a term would be ‘0’ in figure ??.
- C^S . An example of a constant giving a set would be ‘{}’ in figure ??.
- C^E . An example of a constant giving an expression would be ‘ $m' = 0$ ’ where the constants is ‘=’ from figure ??.

binders There are two subsets of binders in the categories for Z specifications. Binders giving sets B^S , and binders giving expressions B^E .

- B^E . An example of a binder giving an expression is

‘ $\exists schedule : TIMESLOT \leftrightarrow ROOM \bullet (allPairsModuleTT \cap schedule = \emptyset \wedge moduleTT = moduleTT \oplus m? \mapsto schedule)$ ’

taken from Timetable specification in [?].

- B^S . An example of a binder giving a set is

‘ $\bigcup\{s : \text{dom studentTT} \bullet \{s \mapsto (\text{studentTT } s \setminus \text{moduleTT } m?)\}$ ’

taken from Timetable specification in [?].

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

- $C^T(\vec{\mathcal{P}})$. An example of a constant giving a term is ‘#n’ (taken from figure ??) the constant being # which would be in the preface of constants with the weak typing as $S \rightarrow T$ and the parameter of this constant giving a term would be ‘n’ which is set.
- V^T . See section ?? on variables giving terms.

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

- $C^S(\vec{\mathcal{P}})$. An example of a constant giving a set with parameters is ‘ $studentTT \cup s? \mapsto \emptyset$ ’

taken from Timetable specification in [?]. Where the constant giving a set is ‘ \cup ’ and the parameters it takes is ‘ $studentTT$ ’ and ‘ $s? \mapsto \emptyset$ ’.

- $B_Z^S(E)$. An example of a binder giving a set with an expression and declaration as parameters is

‘ $\bigcup\{s : \text{dom } \textit{studentTT} \bullet \{s \mapsto (\textit{studentTT } s \setminus \textit{moduleTT } m?)\}\}$ ’

taken from Timetable specification in [?]. Where the constant is ‘ \bigcup ’, the declaration parameter is ‘ $s : \text{dom } \textit{studentTT}$ ’ and the expression parameter is ‘ $\{s \mapsto (\textit{studentTT } s \setminus \textit{moduleTT } m?)\}$ ’.

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

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

- $C^{\mathcal{E}}(\vec{\mathcal{P}})$. A constant giving an expression can be seen in figure ?? as ‘ $m \geq 0$ ’ where ‘ \geq ’ is the constant giving an expression and the parameters are two terms: ‘ m ’ and ‘ 0 ’.
- $B_{\mathcal{Z}}^{\mathcal{E}}(\mathcal{E})$. A binder giving an expression could be a ‘ \forall ’ or ‘ \exists ’ binder. An example of this is shown in the Timetable specification in [?] as

‘ $\exists \textit{schedule} : \text{TIMESLOT} \leftrightarrow \text{ROOM} \bullet (\textit{allPairsmoduleTT} \cap \textit{schedule} = \emptyset \wedge \textit{moduleTT} = \textit{moduleTT} \oplus \{m? \mapsto \textit{schedule}\})$ ’

where the binder giving an expression is ‘ \exists ’, the declaration parameter is ‘ $\text{TIMESLOT} \leftrightarrow \text{ROOM}$ ’ and the binding expression is ‘ $(\textit{allPairsmoduleTT} \cap \textit{schedule} = \emptyset \wedge \textit{moduleTT} = \textit{moduleTT} \oplus \{m? \mapsto \textit{schedule}\})$ ’.

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 parameters giving a set, $C^{\mathbb{S}}(\vec{V}) := \mathbb{S}$. An example of this is shown the GenDB specification in [?]. The definition is

‘**let** $\textit{cosrel} == (\textit{parent}^{nth?+1} ; (\textit{parent}^{-1})^{nth?+1+rem?}) \setminus (\textit{parent} ; \textit{parent}^{-1}) \bullet$
 $\textit{cousins!} = \textit{cosrel}(\{p?\}) \cup \textit{cosrel}^{-1}(\{p?\})$ ’

where the defined constant is \textit{cosrel} .

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

- \emptyset . The empty schema text is the beginning of a specification where we start with nothing.
- Γ, \mathcal{Z} . The first declaration in a specification would be the empty Γ plus the declaration. For example in figure ?? 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.
- Γ, \mathcal{E} . This set represents all the expressions which are added to the schema text. In the example in figure ?? 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 $\Gamma \triangleright \mathcal{E}$ or a definition $\Gamma \triangleright \mathcal{D}$, relative to a schematext.

The symbol \triangleright is a separation marker between the schematext and expression or definition

- $\Gamma \triangleright \mathcal{E}$. Examples of expressions in a paragraph see section ??.
- $\Gamma \triangleright \mathcal{D}$. Example of definitions in a paragraph see section ??.

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

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

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

Declarations. A declaration in a schematext represents the *introduction of a variable* 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^S is a set $V^S : SET$, or term V^T is in the set S , $V^{T:S}$.

- $V^S : SET$. An example of this kind of declaration is ‘ $n : \mathbb{P}N$ ’ taken from figure ??.
- $V^{T:S}$. An example of this kind of declaration is ‘ $m : N$ ’ taken from figure ??.

Parameters. The list of parameters represent the categories in which constants may depend. The parameters available in the ZCGa abstract syntax are terms T , sets S or expressions \mathcal{E} .

4.1.2 Weak Typing Rules

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

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

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 $\text{prefcons}(spec)$ ¹. This set contains all the constants listed in the preface. If $c \in \text{prefcons}(spec)$ and if K_1, \dots, 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 attach the type $k_1 \times \dots \times k_n \rightarrow l$ to c .

Example 4.1.1.

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

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

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

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}	$<$	$\mathcal{T} \times \mathcal{T} \rightarrow \mathcal{E}$
$-$	$\mathcal{T} \times \mathcal{T} \rightarrow \mathcal{T}$	\cup	$\mathbb{S} \times \mathbb{S} \rightarrow \mathbb{S}$

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 $c(x_1, \dots, x_n) := A$. Then the defined constant of the definition, or $defcons(\mathcal{D})$, is c .

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 [?], 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 ??.

We use 7 categories, **Spec**, Γ , \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 ??).

4.2 Annotations

Using the ZMathLang L^AT_EX 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 ?? shows how to label specifications with ZCGa.

Category	L <small>A</small> T <small>E</small> X label	Colour
Specification	<code>\specification{...}</code>	■
SchemaText	<code>\text{...}</code>	■
Term	<code>\term{...}</code>	■
Set	<code>\set{...}</code>	■
Declaration	<code>\declaration{...}</code>	■
Expression	<code>\expression{...}</code>	■
Definition	<code>\definition{...}</code>	■

 Table 4.3: ZCGa LATEX annotations and their colours.

We have chosen these categories in order to label with the specification with minimum amount needed in order to check for weak type correctness. The weak types `term`, `set`, `declaration` and `expression` can be used in a semi-formal specification. An example of this is shown in section ???. At this early stage we wanted to weak-type check our specification and strongl-type checkers already exists, and the specification will be strongly typed when translated to Isabelle anyway. This weak type check will also allow us to check semi-formal specifications.

4.2.1 term

According to the rules in table ?? for an element to be a well typed term it can be a variable being declared such as `t` (labelled in blue) in figure ?? 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 ??).

<code>\term{\# \set{s}}</code>	# s
--------------------------------	---

Figure 4.2: Constant giving a term

Figure ?? 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 typed in one of two ways. The first is a variable set which is correctly declared such as ‘ s ’ in figure ???. 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 ?? 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 its type. For example if we had the declaration $t : \text{nat}$ this is declaring t is of type some sort of natural number. We can label this declaration in ZCGa (shown in figure ??)

$\declaration{\term{t}}{\expression{\text{nat}}}$	$t : \mathbb{N}$
---	------------------

Figure 4.4: Correct term declaration labelled in zcg

We label \mathbb{N} as declarations for the same reasons as the typing’s behavior in [?].

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

$\declaration{\set{s}}{\expression{\text{power } \text{nat}}}$	$s : \mathbb{P}\mathbb{N}$
--	----------------------------

Figure 4.5: Correct set declaration labelled in zcg

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 ??.

<code>\expression{\term{t} \in \set{s}}</code>	<code>t ∈ s</code>
--	--------------------

Figure 4.6: Correct expression labelled in zcga

4.2.5 definition

A correct definition is if all the variables within the definition have been declared and the constant the user is defining is a constant set and the constant is not already in predefined or the preface constants. A correct Z definition is shown in figure ??.

<code>\definition{\LET \set{old} == \set{new} @</code>	
<code>\expression{\term{t} \in \set{old}} \definition{\LET \set{old} == \set{new} @</code>	
<code>\expression{\term{t} \in \set{old}}</code>	
<code>let old == new • t ∈ old</code>	

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 ?? 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}
```

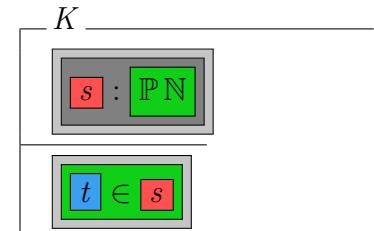


Figure 4.8: Example of correct schematext

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 ?? in figure ??.

4.3 Implementation

The ZCGa program automatically checks the specification for grammatical correctness. To do this it uses the ZCGa annotations are 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 if 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, 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 ??.

If the specification is correctly labelled and follows all the rules in table ?? 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 ?? 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 ??). 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 ???. 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 previously 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 ???. The message *set not declared* will appear when a variable is labelled `\set{..}` in the schematext of a specification 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 L^AT_EX 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 numbers `\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 [STUDENTS] 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 used 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 ZCGa on a semiformal specification

The ZCGa can also be used on semiformal specification. An example is shown in appendix ???, 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 grammatical 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 ??). 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.

1. 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:

- 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] | [press_cas_eng] | [press_alt_eng] |
[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 `alt_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] | [engaged]
```

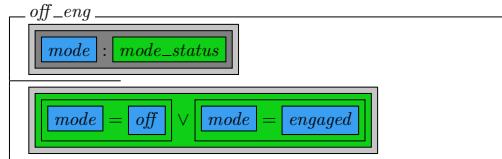


Figure 4.10: Part of the Autopilot specification labelled in ZCGa.

4.4.1 Semi-formal specification is ZCGa incorrect

One main advantage of the ZCGa is that a semi-formal specification can be checked for weak type correctness. This is useful during times when a specification for a system is written by many people which is usually the case.

1. 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:

- 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 | press_cas_eng | press_alt_eng |
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 `alt_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 | engaged
```

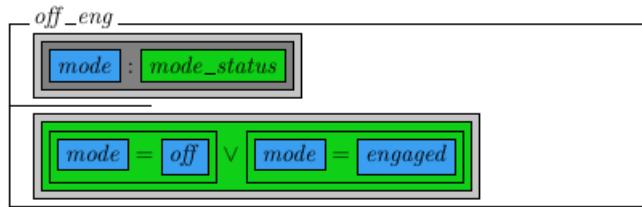


Figure 4.11: Part of the Autopilot specification labelled in ZCGa but with terms within the informal text.

Figure ?? shows part of the semi-formal autopilot specification but the user has annotated the terms within the informal part of the specification. This annotation is legal in ZMathLang.

If we run this specification through the ZCGa checker the output message will show that there are 4 errors and the terms `att_cws`, `fpa_sel`, `alt_eng` and `cas_eng` have not been declared. This is shown in figure ??.

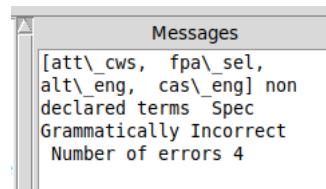


Figure 4.12: Output result of running passing Figure ?? through the ZCGa checker.

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

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 ??).

4.5 ZCGa on an informal specification

If multiple people are working on one specification, one might have the following issue which ZCGa will help to identify.

System Engineer A writes the following requirements:

1. The vehicle shall stop at a red light
2. The vehicle shall not kill any human being
3. The vehicle shall not harm any human being
4. The vehicle shall hand over control to the driver when requested
5. A **driver** : *shall be a human being over the age of 18*
sitting inside the vehicle
6. etc, etc

System engineer B writes the following set of requirements:

1. The driver shall be in sitting in a comfortable position in the vehicle.
2. The **driver** : *shall be a human being over the age of 17*
sitting inside the vehicle
3. The driver shall be relaxed
4. etc, etc

In this case the ZMathLang toolkit will flag up an error because requirement 5 from A conflicts with requirement 2 from B. The error message for these requirements are shown in figure ??.

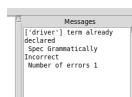


Figure 4.13: Output result of running passing informal requirements through the ZCGa checker.

4.6 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 members 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 a partially-formal specifications. These can include specifications written in natural language but are on their way to becoming formal, or specifications with formal parts to them.

We have also shown a benefit of using the ZCGa weak type checker on informal requirements where a term is declared twice by 2 different people. The ZCGa will also catch if a set has been declared more than once in a specification.

4.7 Conclusion

In this chapter we have seen how the ZCGa has grown from weak type theory for mathematics [?]. We have given 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

²As 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

how a Z specification can be annotated and checked by 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 in MathLang. Here we describe how the ZDRa was designed and implemented.

We use the ZMathLang L^AT_EX 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).

We have chosen to identify loops in the reasoning as it determines whether the program is deterministic. Safety-critical systems must be deterministic, practices

which can lead to non-determinism must be avoided or carefully controlled [?]. Identifying loops in the reasoning is one of these practices. Checking for loops in the reasoning can also be automatically checked on an informal specification as shown in section ??.

There are millions of safety properties one can check for however we have chosen to check the state invariants & checking the robustness of the specification as these will be the easiest to automate using the information we have from the annotations.

It is important to have a robust and totalised specification (in particular one for a high integrity system) because totalisation leaves less room for error. Specifications which are totalised are complete and therefore system requirements are better understood and the programmer would have less assumptions about the system. By totalising schemas all preconditions will have a corresponding postcondition. Therefore when executing the system if all possible preconditions have a corresponding output (post condition) then the computing system will be able to handle errors better. Building a robust system encompasses as many points of failure as possible and one way to do this is to make the specification total.

Identifying relationships between chunks of specification is useful as it allows us to carry out computations whether all dependencies are identified and to check whether the relationships are sensible (for example all pre conditions have a corresponding postconditions, a schema is defined if is used in another schema). The relationships also allow users to explain the logical structure of their system specification.

State invariants are conditions that can be relied upon to be true during the execution of the program. It is important that state invariants do not change as they confirm your system specification is valid if the state invariants hold even as the user states the preconditions and post conditions of each schema.

The dependency graph is useful as it allows the user to identify which chunks of specification are '*dependent*' on each other i.e. if one chunk was changed or removed then what other chunks would be affected. The dependency graph also allow us to build the GoTo graph which orders the specification to be automatically written in

a theorem prover.

Section ?? describes the labels used to annotate a specification. Then in section ?? 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 ?? and ?? and the products which are created when a specification is rhetorically correct are described in section ??.

5.1 Annotations

In a similar way as MathLang for mathematics ??, we can label the specifications with ZDRa annotations (either before or after the ZCGa) using our ZMathLang L^AT_EX package . These annotations chunks parts of the specification together and upon compiling shows the relationships between each of these chunks.

We have chosen the following annotations for similar reasons as the ZCGa. We only wanted the minimum amount of categories in order to check for rhetorical correctness. We also wanted the minimum amount of categories to translate the specification into an Isabelle skeleton.

Instance	Notation	L <small>A</small> T <small>E</small> X Command
theory	T	$\backslash dratheory\{T\}$ $\{scaleoftheory\}\{instance\}$
stateschema	SS	$\backslash draschema\{SS\#\}\{instance\}$
initschema	IS	$\backslash draschema\{IS\#\}\{instance\}$
changeschema	CS	$\backslash draschema\{CS\#\}\{instance\}$
outputschema	OS	$\backslash draschema\{OS\#\}\{instance\}$
totaliseschema	TS	$\backslash draschema\{TS\#\}\{instance\}$
axiom	A	$\backslash draschema\{A\#\}\{instance\}$
stateInvariants	SI	$\backslash draline\{SI\#\}\{instance\}$
precondition	PRE	$\backslash draline\{PRE\#\}\{instance\}$
postcondition	PO	$\backslash draline\{PO\#\}\{instance\}$
output	O	$\backslash draline\{O\#\}\{instance\}$

 Table 5.1: ZDRA instances with their notations and LATEX commands.

Relation	L <small>A</small> T <small>E</small> X Command
initialOf	$\backslash initialof\{instance_1\}\{instance_2\}$
uses	$\backslash uses\{instance_1\}\{instance_2\}$
requires	$\backslash requires\{instance_1\}\{instance_2\}$
allows	$\backslash allows\{instance_1\}\{instance_2\}$
totalises	$\backslash totalises\{instance_1\}\{instance_2\}$

 Table 5.2: ZDRA Relations with their notations and LATEX commands.

Table ?? 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 ?? shows the relationships which are available between some of the instances.

The instances described in table ?? are the building blocks of what makes a Z specification. These large chunks of specification are what were common across

the literature describing Z specifications including Ed Curries descriptions [?] and Spiveys descriptions [?] in one form or another. Just like in MathLang for mathematics ?? which included instances such as Lemma, Theorem, Proof, Definitions we can group chunk of text together to identify what is a theory, stateschema, initschema, changeschema, outputschema, totaliseschema, axiom, stateInvariants, precondition, postcondition and output.

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.

If we combine together *changeschema* and *outputschema* then these instances are the minimum amount of labels we need to create a compilable Isabelle file as both *changeschema* and *outputschema* become ‘*definitions*’ in Isabelle. However we have decided to label *changeschema* and *outputschema* separately as the user can easily identify which schemas can change the state of the system and which just output particular results.

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 ?? 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.

Stateschema's are important to identify as they go in the premuable of the Isabelle file.

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 ?? 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{P02}{known' = \{\ \}}
  \end{schema}}
```

Figure 5.2: An *initialschema* labelled in ZDRA.

Initschema's are important to identify as they become *definitions* Isabelle file (if it exists as they are not always defined in specifications).

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.

¹Although it is not needed for the user to annotate the instances incrementally (the instances just need different names) we have used it in our example to make it easier for the reader to identify how many of each instance is within the specification.

```
\draschema{CS1}{  

\begin{schema}{AddBirthday}  

\Delta BirthdayBook \  

name?: NAME \  

date?: DATE  

\where  

\dra{PRE1}{name? \notin known}  

\dra{P03}{birthday' = birthday \cup \{name? \mapsto  

date?\}}  

\end{schema} }
```

Figure 5.3: A changeschema labelled in ZDRa.

Figure ?? 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.

Changeschema's are important to identify as they become *definitions* Isabelle file (if it exists as they are not always defined in specifications).

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 ?? 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{04}{result! = already\_known}  

\end{schema} }
```

Figure 5.4: A outputschema labelled in ZDRA.

Outputschema's are important to identify as they become *definitions* Isabelle file (if it exists as they are not always defined in specifications).

5.1.1.6 totaliseSchema

Totaliseschema instances are parts of the specification which totalise preconditions within the specification. These are labelled as TS# where # is some number. *Totaliseschema* 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 *totaliseschema* 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 *totaliseschema* 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 ?? 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 ???. In this example we have three *totalise instances* using the label `\draline{TS1}{...}, \draline{TS2}{...}` and `\draline{TS3}{...}` respectfully (shown in red).

Totaliseschema instances become the properties to prove when converted to the half-baked proof (see chapter ?? for details). This is because totaliseschema instances will should contain a preconditions and postconditions which should obey they stateinvariants within the specification. Totaliseschema instances should be total and therefore complete. If all the totaliseschemas are proved correct then their should be less room for errors within the entire system.

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 ???. 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.

Axioms are important to identify as they go in the premuable of the Isabelle file.

5.1.1.8 stateInvariants

The *stateInvariants* instance are the conditions which must be obeyed throughout the specifications. These are the lines found inside the *stateschema* instance. Figure ?? shows a single *stateinvariant* instance labelled as \draline{SI1}{...} (shown in blue). There can be 0 or more *stateinvariance* instances within a *theory*. *StateInvariants* are important to be identified in the ZDRA as they come after the ‘*assumes*’ in the isabelle syntax (see chapter ?? to see how stateinvariants are translated). Each definition and lemma then described in Isabelle makes sure that all conditions obey the stateinvariants.

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 ?? and ???. In figure ?? 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 ?? 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 ???. The ZDRA labelling consists of the line `\draschema{PRE1}{..}` where the name of the instance is PRE1.

Precondition's are important to identify as they must be checked that they conform with the stateInvariants of the specification.

5.1.1.10 postcondition

The *postcondition* instance can be labelled as a ZDRA line. An example of this is demonstrated in figure ?? (shown in green), where the name of the instance is P03 and the instance itself is `birthday' = birthday \cup \{name? \mapsto date?\}`.

Postconditions's are important to identify as they must be checked that they conform with the stateInvariants of the specification.

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 ???. The instance in this case is named 04 and the instance itself is `result! = already_known`.

Output's are important to identify as they must be checked that they conform with the stateInvariants of the specification.

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 → precondition	outputSchema → stateSchema
outputSchema → output	changeSchema → stateSchema
changeSchema → precondition	stateSchema → stateSchema
changeSchema → postcondition	stateSchema → axiom
totalises	outputSchema → axiom
totaliseschema → changeSchema	changeSchema → axiom
totaliseschema → outputSchema	allows
totaliseschema → totaliseschema	precondition → postcondition
totaliseschema → precondition	initialOf
	initialSchema → stateSchema

Table 5.3: The legal relations between instances. Where → represents the relation.

Table ?? 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. These relations are all we need to create a Dependency and GoTo graph and to describe the relationships between the instances of the specification. These relations are also the minimum we need to create the program to automatically translate the specification in the correct order into an Isabelle file.

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 ?? we can say that the *initialSchema* IS1 *requires* the postcondition P02. Therefore in ZDRA notation we would write in our L^AT_EX specification \requires{IS1}{P02}.

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 ??, where we have a *precondition* PRE3 (in blue) and an *output* O4 (in green). In this case the user would write in their ZDRA L^AT_EX file: \allows{PRE3}{O4}.

5.1.2.5 totalises

The *totalise* relation allows users to describe which of their schema's have been totalised. If there exists a precondition which has not been totalised than the ZDRA check will identify this (see next section). Any schema which contains *preconditions* can be totalised in a *totaliseschema*. For example the the user can label \totalises{TS1}{PRE1}

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 L^AT_EX 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.

It is important to have correctly totalised specifications, especially in safety critical systems as it leaves less room for error when translating specification into actual programs. If every preconditions have a corresponding postcondition than the specification will be deemed '*totalised*' and therefore a safer system. With a totalised specification the programmer would not need to make any assumptions and any defined state the system is in will have a corresponding postcondition. Totalisation also adds a level of rigour to the specification for example in the Sholis project [?] described in chapter ?? the Z specification found 75% of faults in the system. Without a totalised specification those faults would of been implemented in the system and then potentially only found during testing where a lot of time and money would of been spent.

5.2.1.1 Errors

Table ?? 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 incorrectly totalised’ appears. Specification 3 in table ?? 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}	‘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 its 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.

Loops in the reasoning identify when two specifications or functions can call one another (mutual recursion). This should be avoided when writing safety critical systems because a system which has mutual recursion could possibly overload the stack and the software will crash.

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 $[a \rightarrow b, a \rightarrow c, b \rightarrow c]$ then that would still be legal as there are no directed loops, however, if there was a graph with edges $[a \rightarrow b, b \rightarrow c, c \rightarrow a]$ then that would cause a loop in the reasoning and the specification will not be ZDRA correct. The program outputs a message informing the user whether the specification is ZDRA correct or not.

```
... \draschema{SS1}{

  \begin{schema}{A}
  C
  \end{schema}

  \draschema{SS2}{

    \begin{schema}{B}
    A
    \end{schema}

    \end{schema}

    \draschema{SS3}{

      \begin{schema}{C}
      B
      \end{schema}

      \end{schema}

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

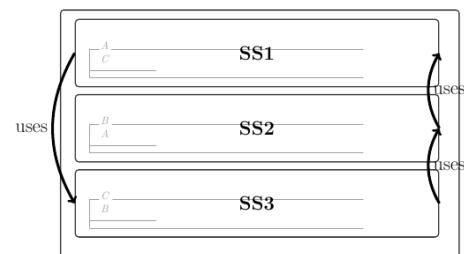


Figure 5.10: The pdflatex output of figure ??.

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

Figure ?? 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 ??

```
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 ??). If not then the graphs are not created.

A specification which passes ZCGa but not ZDRA is shown in appendix ??.

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 ?? and the full version is shown in appendix ??.

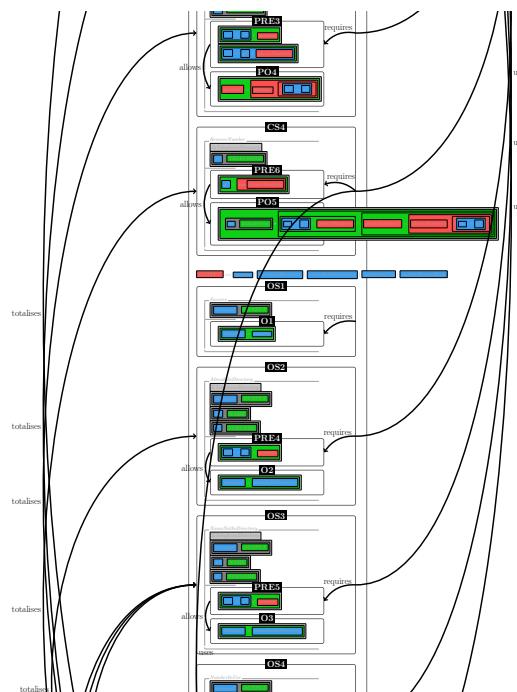


Figure 5.12: A snippet from appendix ?? showing a loop in reasoning.

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 is shown in figure ?? in appendix ??.

5.2.3 ZDRa Outputs

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 directed graph built within the ZDRa check (see section ??). Further information on how these products are formed can be found in the next chapter.

5.2.3.1 Dependency Graph

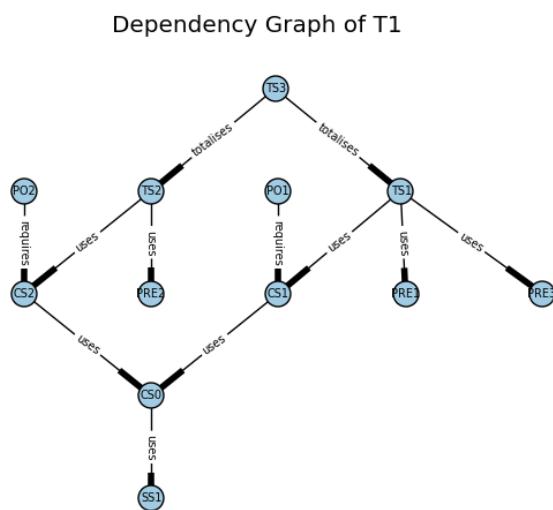


Figure 5.13: An example of a dependency graph.

An example of a *dependency graph* can be seen in figure ???. This image represents the compiled ZDRa annotated document in 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.

The dependency graph is a directed graph representing the dependencies of instances. The benefits of a dependency graph would be for system analysis, one can see which parts of the specification are **dependent** on others. For example if one were to change a function (schema) which other functions would be affected. Circular reasoning will also be easy to identify on a dependency graph.

5.2.3.2 GoTo Graph

GoTo graph of T1

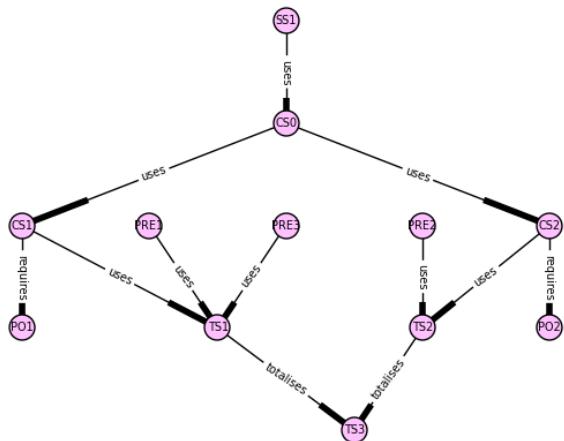


Figure 5.14: An example of a goto graph ZDRa correct.

An example of a *goto graph* is shown in figure ??, 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 initialize it. Therefore 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*.

The GoTo graph represents the order the instances should go in when translating the specification into Isabelle syntax.

The dependency and GoTo graphs contain the same instances and the same relations, however the direction of the relations may be different. These differences between the dependency and GoTo graphs are described in the next chapter.

5.3 ZDRa on an Informal Specification

One of the main benefits is that the ZDRa checker can be used on specifications which are written entirely in natural language. This is particularly useful if the specification has been written by multiple people.

For example the specification below shows a list of requirements.

1. The Unmanned Air Vehicle takes command from the Wildcat Helicopter.
2. The Unmanned Air Vehicle shall stay in it's allocated airspace.
3. The Wildcat Helicopter takes command from the Chinook.
4. The Wildcat Helicopter shall stay on base until told to deploy.
5. The Chinook takes command from the Unmanned Air Vehicle.

The user can then annotated these informal requirements in ZDRa. This is shown in figure ?? where the ZDRa commands are in blue.

```
\documentclass{article}
\usepackage{zmathlang}
\begin{document}
\dratheory{T5}{0.3}{}
\draschema[CS1]{The Unmanned Air Vehicle takes command from the Wildcat Helicopter.}
\draschema[CS2]{The Unmanned Air Vehicle shall stay in it's allocated airspace.}
\draschema[CS3]{The Wildcat Helicopter takes command from the Chinook.}
\draschema[CS4]{The Wildcat Helicopter shall stay on base until told to deploy.}
\draschema[CS5]{The Chinook takes command from the Unmanned Air Vehicle.}

\uses{CS1}{CS5}
\uses{CS5}{CS3}
\uses{CS3}{CS1}
\end{document}
```

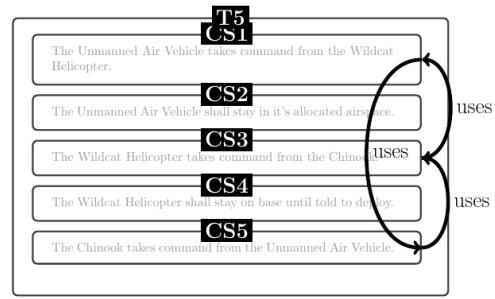


Figure 5.15: Informal requirements example ZDRA.

The ZMathLang toolkit will identify the circular reasoning and tell the user (requirements 1, 3 and 5). It will also produce the following graph to show that the 3 requirements are dependent on each other and therefore show there is an error. This loop is shown in figure ??.

Dependency Graph of T5

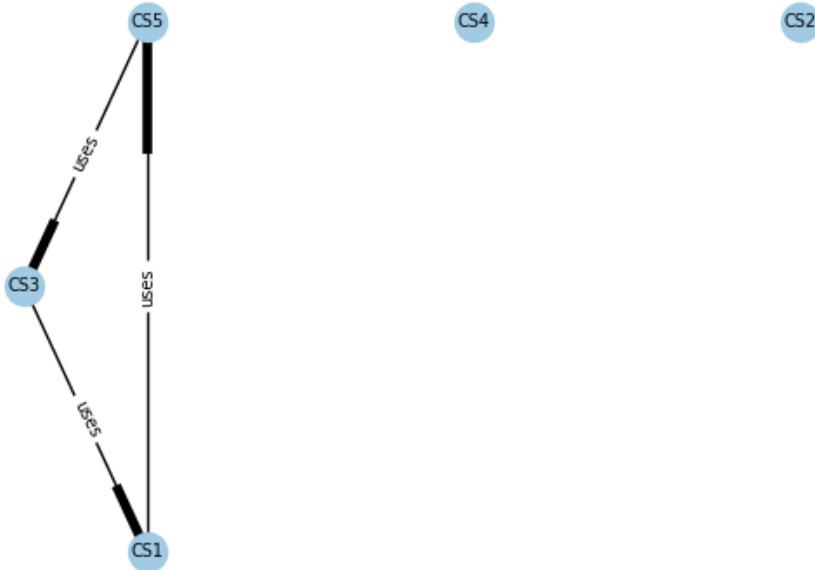


Figure 5.16: Dependency graph of the loop in the informal specification example detailed in figure ??.

5.4 Conclusion

In this chapter the ZDRa step of the ZMathLang has been described. A L^AT_EX 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. We have also formally outlined the ZDRa and shown some rules which occur when combining the ZCGa and ZDRa together. 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 Proof Sketch

The proof sketches described in this chapter are automatically generated if the specification passes the ZDRa check. Section ?? outlines how the dependency and goto graphs are generated from the ZDRa annotations. Then section ?? describes how ZMathLang builds the Gpsa from the graphs. We describe how proof obligations are generated from the proof sketch. Proof obligations are logical formulas associated to a correctness claim for a given verification property. Therefore by generating proof obligations from the General Proof Skeleton aspect (GPSa) we can then establish some correctness of the specification as they check for consistency and sanity of the specification.

6.1 Building the Dependency and GoTo graphs

In this section we outline how the dependency graph is created from the ZDRa annotations. We see that each relation becomes an ‘edge’ in the graph and each instance becomes a ‘node’ in the graph. The goto graph is then built from the dependency graph with the same nodes but some edges being reversed.

6.1.1 How ZMathLang builds the dependency graph from the ZDRa annotations

We use the definitions from [?] to describe ‘nodes’ ‘relational edges’ to describe the dependency graph as a set.

Definition 6.1.1 (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 6.1.2 (Relational Edges, [?]). *We can say a relation edge \mathbb{E} , is a triple*

$$(v_1, rel, v_2)$$

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 6.1.3 (Dep graph set, [?]). *A dependency graph \mathbb{D} , is a set of relational edges \mathbb{E} , and a set of nodes \mathbb{V} .*

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

Using the definitions from [?], we can group the nodes and edges together to describe the ZDRa annotations.

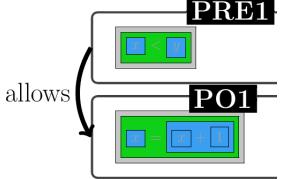
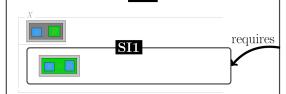
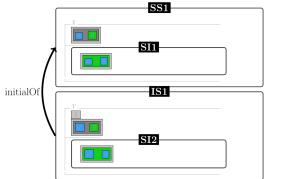
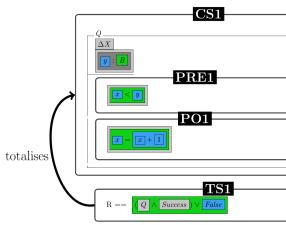
Nº	Annotations	
dra1)		$\mathbb{V}_{dra1} = \{(PRE1, precondition), (PO1, postcondition)\}$ $\mathbb{E}_{dra1} = \{((PRE1, precondition), allows, (PO1, postcondition))\}$ $\mathbb{D}_{dra1} = \mathbb{V}_{dra1} \cup \mathbb{E}_{dra1}$
dra2)		$\mathbb{V}_{dra2} = \{(SS1, stateSchema), (SI1, stateInvariants)\}$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

Table 6.1: Using the formalised definitions for vertices and edges to create a dependency graph.

Definition 6.1.4 (ChildOf). *The dependency graph provides a child and parent*

relationship. The definitions are as follows:

Relation	dependency graph children
\allows{A}{B}	B childOf A
\requires{A}{B}	B childOf A
\uses{A}{B}	B childOf A
\initialOf{A}{B}	B childOf A
\totalises{A}{B}	B childOf A

The goto graph also provides and child and parent relationship however some of

the relations are reversed:

Relation	goto graph children
\allows{A}{B}	B childOf A
\requires{A}{B}	B childOf A
\uses{A}{B}	A childOf B
\initialOf{A}{B}	A childOf B
\totalises{A}{B}	A childOf B

Table ?? 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 *initialOf* 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 ??). If the *allows* relationship is used within the schema, then both the precondition and postcondition/output becomes childrenOf the schema. Figure ?? shows two separate schemas which are not nested within each other, however figure ?? shows the precondition PRE2 allows the postcondition PO2 within the schema CS1, since CS1 requires PRE2 then both PRE2 and PO2 are childrenOf CS1.

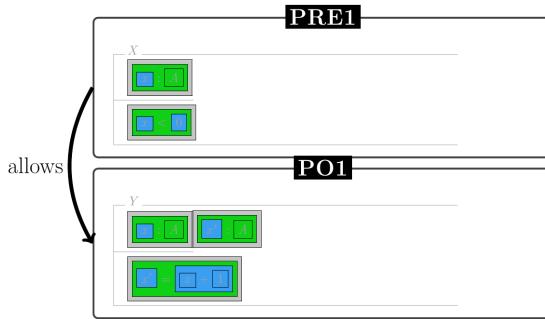


Figure 6.1: Relation with un-nested precondition and postcondition.

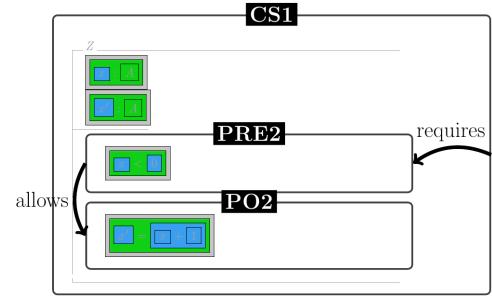


Figure 6.2: Relation with nested precondition and postcondition.

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

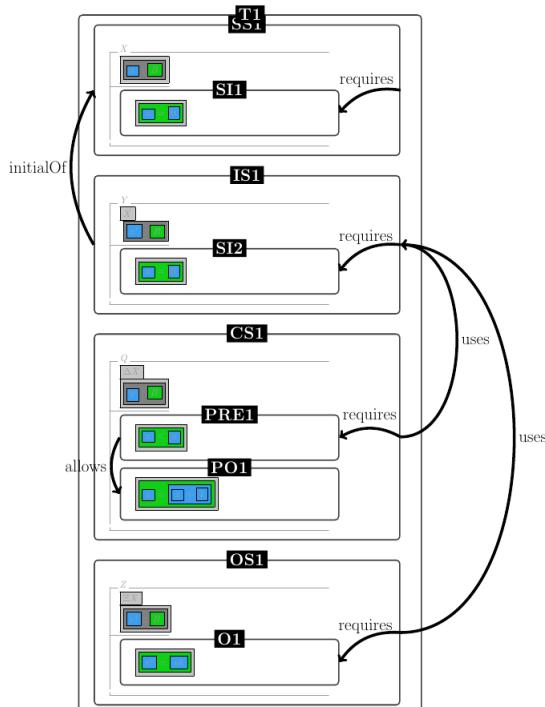


Figure 6.3: All annotations from table ?? combined into one specification.

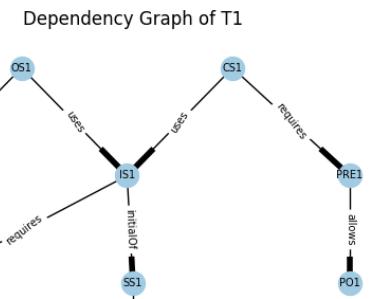


Figure 6.4: Dependency graph of the example in ??

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

$$\begin{aligned} \mathbb{V}_{comb} = & \{ \\ & (T1, theory), (SS1, stateSchema), (SI1, stateInvariants), \\ & (IS1, initSchema), (SI2, stateInvariants), (CS1, changeSchema), \\ & (PRE1, precondition), (PO1, postcondition), (OS1, outputSchema), (O1, output) \} \end{aligned}$$

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

6.1.2 Generating the GoTo graph from the dependency graph

Since the dependency graph only follows the annotations of the ZDRa we need to change some of the ordering of some of the relations for the GoTo graph. Zengler [?] uses *textual orders* to change the order for mathematical relations (strong textual order, weak textual order, common textual order), however for Z specifications we need to change the ordering as shown in the algorithm ???. When inputting a specification into an automatic theorem prover, (in this case Isabelle) the order is important as it decides which part of the specification needs to be in-putted first in

order to parse correctly.

```

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_2, →, v_1) 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 1: Algorithm to generate the dependency graph and goto.

Algorithm ?? 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 draschema 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 ?? 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.

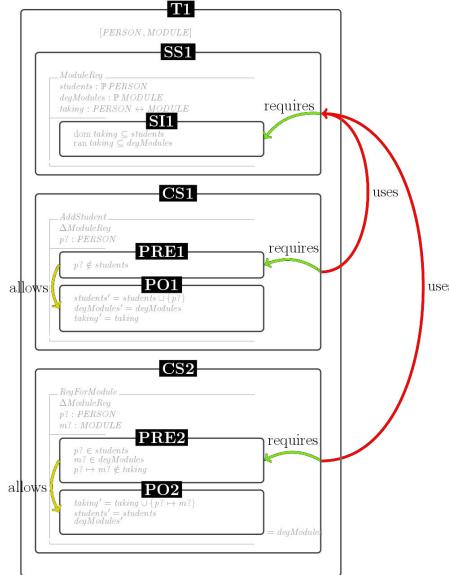


Figure 6.5: User annotated in ZDRA for the modulereg specification with arrows coloured.

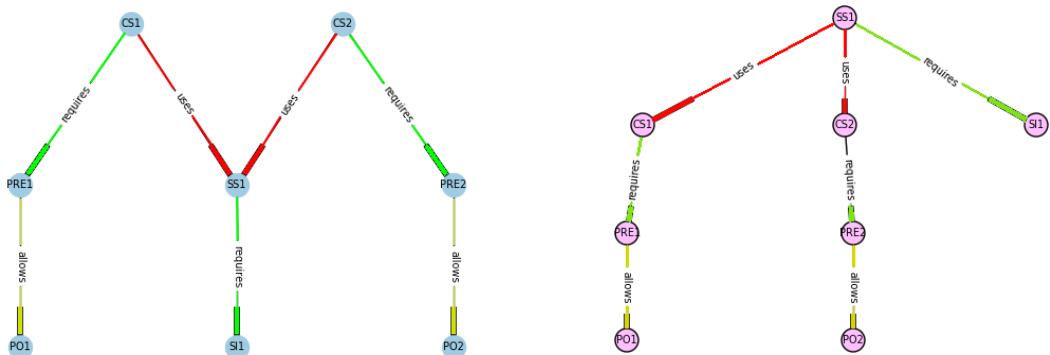


Figure 6.6: The dependency graph of modulereg specification with arrows coloured.

Figure 6.7: The goto graph of modulereg specification with arrows coloured.

Figure ?? 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 ??) and its corresponding goto graph (figure ??). 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.

The main reason for producing a GoTo graph for a Z specification is to order the instances so that when printed they can parse through the Isabelle theorem prover. However both the dependency and goto graphs can also be used as documents to analyse the system in question. It may also be helpful to stakeholders in the systems project to view the system in it’s entirety and to see clearly how each component is linked with another.

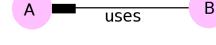
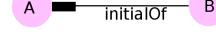
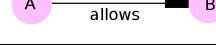
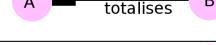
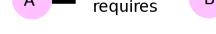
Relation	GoTo Edge
(v_A, uses, v_B)	
$(v_A, \text{initialOf}, v_B)$	
$(v_A, \text{allows}, v_B)$	
$(v_A, \text{totalises}, v_B)$	
$(v_A, \text{requires}, v_B)$	

Table 6.2: The relations represented by textual order and in the goto graph

Table ?? shows what the order representation of each relation is and how it is represented as an edge in the GoTo graph. The uses relation has node b going towards node a therefore node b will need to be written in Isabelle before node a . The initialOf relation is similar and will require node b to be written in Isabelle before node a and we have $a, \text{initialOf}, b$. The same concept goes for the totalises relation. The allows relation and requires relation are the opposite and keep the same order in the GoTo graph as the dependency graph.

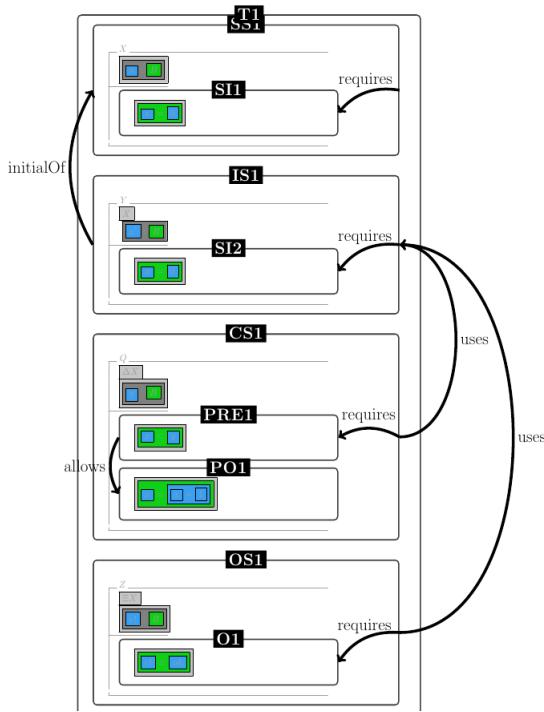


Figure 6.8: All annotations from table ?? combined into one specification.

GoTo graph of T1

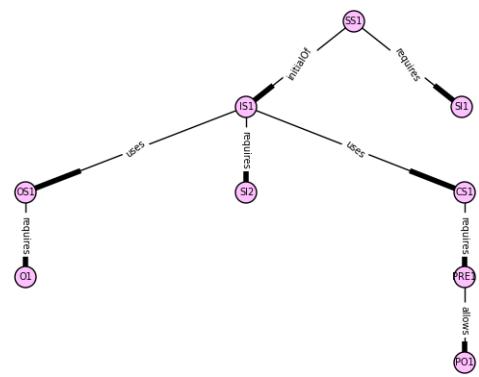


Figure 6.9: Goto graph of the example in ??

Therefore our nodes would be:

$$\mathbb{V}_{\text{figure } ??} = \{ (T1, \text{theory}), (SS1, \text{stateSchema}), (SI1, \text{stateInvariants}), (IS1, \text{initSchema}), (SI2, \text{stateInvariants}), (CS1, \text{changeSchema}), (PRE1, \text{precondition}), (PO1, \text{postcondition}), (OS1, \text{outputSchema}), (O1, \text{output}) \}$$

and our edges would be:

$$\mathbb{O}_{\text{figure } ??} = \{ ((SS1, \text{stateSchema}), (SI1, \text{stateInvariants}), \text{requires}, \leftrightarrow) \\ ((IS1, \text{initSchema}), (SI2, \text{stateInvariants}), \text{requires}, \leftrightarrow) \\ ((IS1, \text{initSchema}), (SS1, \text{stateSchema}), \text{initialOf}, \succ) \\ ((CS1, \text{changeSchema}), (PRE1, \text{precondition}), \text{requires}, \leftrightarrow) \\ ((PRE1, \text{precondition}), \text{allows}, (PO1, \text{postcondition}), \text{allows}, \preceq) \\ ((CS1, \text{changeSchema}), (IS1, \text{initSchema}), \text{uses}, \succ) \\ ((OS1, \text{outputSchema}), (O1, \text{output}), \text{requires}, \leftrightarrow) \\ ((OS1, \text{outputSchema}), (SS1, \text{stateSchema}), \text{uses}, \succ) \}$$

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

6.2 What is a 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 sketch. Algorithm ?? shows psuedocode in generating this proof sketech.

```
#The order of the graph will start with all the nodes which are r
#dependent on anything
    for allnodes in fromnodes:
        if allnodes not in tonodes:
            appendtoset(allnodes, orderofgraph)
#Remove the nodes which are not dependent on anything from the
#set of all nodes
    for thenodes in allNodesInGraph:
        if thenodes in orderofgraph:
            allNodesInGraph.remove(thenodes)
#Loops through all the nodes, if the nodes parents are printed in
#orderofgraph then add the node to the order and remove from the
#set of all nodes
    while allNodesInGraph:
        for k in allNodesInGraph:
            l = set(goto_graph.predecessors(k))
            if l.issubset(set(orderofgraph)):
                appendtoset(k, orderofgraph)
                allNodesInGraph.remove(k)
```

Figure 6.10: Part of the algorithm to create a proof.

Data: instances are known as ‘nodes’

Data: OrderOfGraph is an ordered graph which lists the proof sketch

Data: ListOfAllNodes is an unordered list containing all nodes

```

1 for each node in ListOfAllNodes do
2   if node is a root then
3     push to OrderOfGraph
4   else
5     nothing
6   end
7 end
8 for each node in ListOfAllNodes do
9   if node in OrderOfGraph then
10    remove node from ListOfAllNodes
11   else
12    nothing
13   end
14 end
15 while there are nodes in ListOfAllNodes do
16   for each node in ListOfAllNodes do
17     predes = predecessor of each node;
18     if predes is a subset of OrderOfGraph then
19       push each node to OrderOfGraph;
20       remove each node;
21     else
22       nothing
23     end
24   end
25 end

```

Algorithm 2: Psuedo algorithm to create a proof sketech.

6.3 Creating the Graph

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

6.3.1 Finding the Parents

The first step of translating the GoTo graph into a proof sketch is to find the nodes which do not have any parents. That is, the nodes which do not have any predecessors.

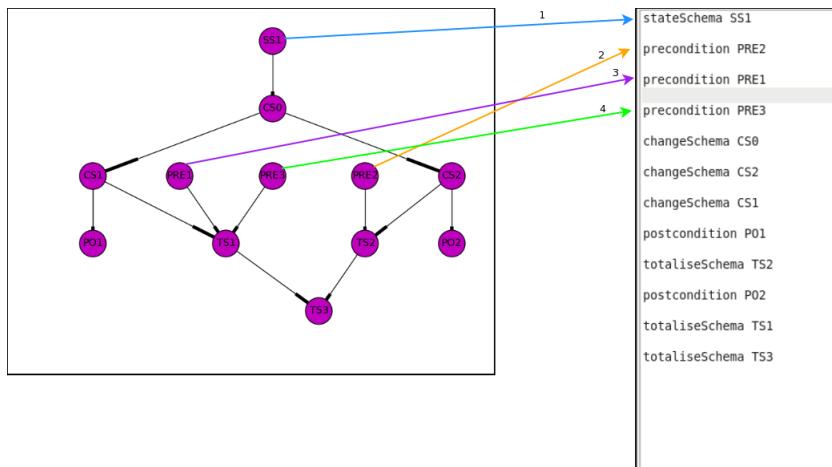


Figure 6.11: GoTo graph and proof sketch of vending machine step 1.

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 figure ?? we see that we have SS1, PRE1 PRE2 and PRE3 all printed.

6.3.2 Children with all level 1 parents

The next step of the algorithm is to find any child nodes which already have all their parents printed. If there exists a node whose parents have no predecessors then these node go next onto the list and become the second level.

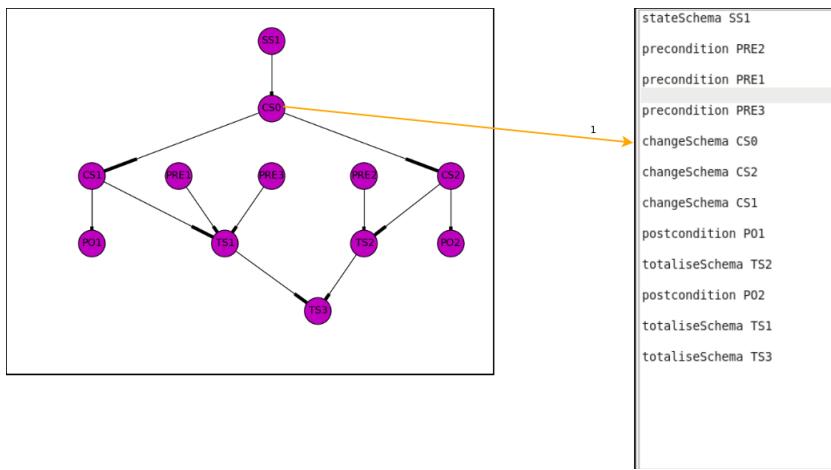


Figure 6.12: GoTo graph and proof sketch of vending machine step 2.

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 sketch. Figure ?? shows that the next node to be in the proof sketch is CS0.

6.3.3 Finding the rest of the children

The following section loops over itself until all of the children have been found. The algorithm looks at the remaining nodes and if their predecessors have already been written onto the proof sketch then it writes that node. For example if node x and node y are parents of node z and node x and node y are already on the proof sketch then it will write node z to the proof sketch and remove it from the list of remaining nodes.

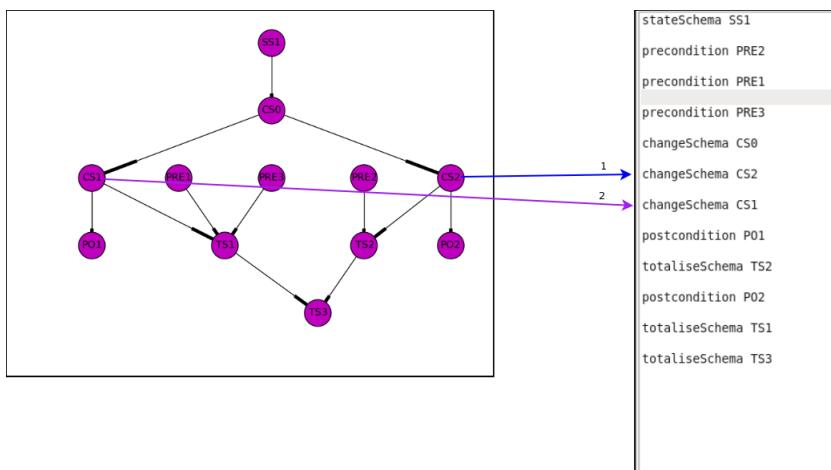


Figure 6.13: GoTo graph and proof sketch of vending machine step 3.

The next part we see that after CS0 is added to the proof sketch then both CS1,

and CS2 can be added. This is shown in figure ??.

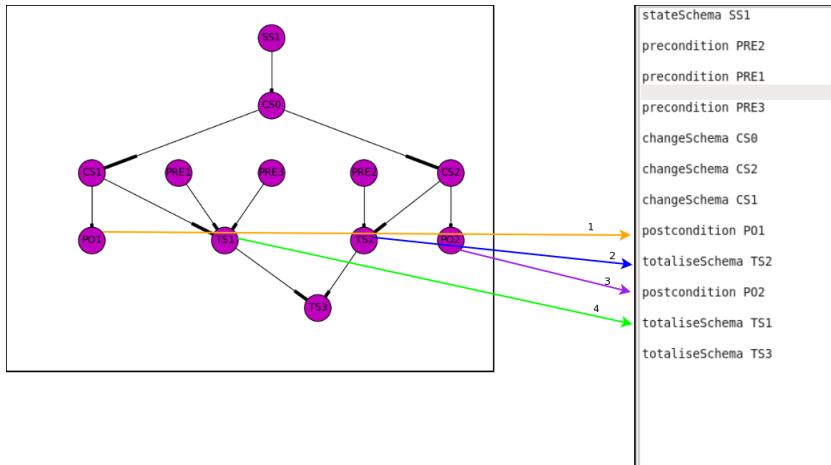


Figure 6.14: GoTo graph and proof sketch of vending machine step 4.

Figure ?? shows the next stage of adding nodes to the Proof sketch. Since CS1 and CS2 are now added to the proof sketch 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 randomly.

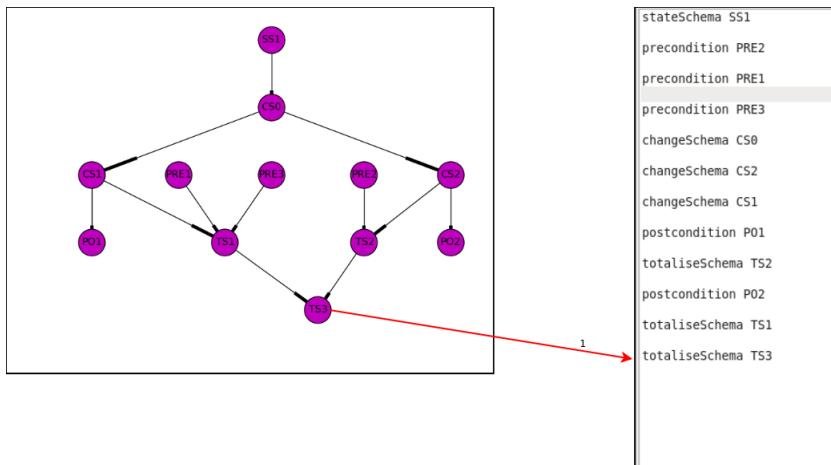


Figure 6.15: GoTo graph and proof sketch of vending machine step 5.

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.

6.4 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 consistency. Using the description in [?], checking the specification is consistent can fall under two categories:

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

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

Out of the many proof obligations to choose we have chosen POb1 as these proof proof obligations would be easiest to automatically generate using the information we have from the ZCGa and ZDRA. We have chosen to add POb2 as these proof obligations are specific to the formal specification and allows the user choice in what they would like to prove.

6.4.1 POb1, Proof Obligation type 1

Definition 6.4.1. *POb1*

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

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

POb1 shows the feasibility of an operation. When an operation can transfer 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 doesn't 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 \<subseteqq> students)
   \<and> (Range taking \<subseteqq> degModules)
   \<and> (Domain taking' \<subseteqq> students')
   \<and> (Range taking' \<subseteqq> 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 states that there exists some variables of the operational schema where the precondition (PRE1) and postCondition (PO1) imply that the stateInvariants (SI1) and stateInvariants prime (SI1'), hold.

6.4.2 POb2, Proof Obligation type 2

As POb2 are any other relevant 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 \<union> {(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.

6.4.3 Proof Obligations in the Proof Sketch

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

The proof obligations which check that changeSchema's and outputSchema's follow the stateInvariants are added to the original proof sketch. The proof sketch 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 sketch. Algorithm ?? shows the algorithm which creates the ordered list of proof obligations. The number

of proof obligations will depend on how many changeSchema's, outputSchemas and totalSchemas there are in the specification.

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, adds 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 the 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 6.16: Example of a Proof Sketch with lemma's.

An example of a Proof Sketch with added proof obligations is shown in Figure ???. 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.

In the next chapter we fill in these proof obligations into predicates and they then become the properties to prove the correctness of the specification.

6.4.3.1 Proof Obligations in specification examples

Since the vending machine specification (appendix ??) doesn't have any stateInvariants then the Gpsa will not have any added proof obligations to check for consistence. 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 ??) does have stateInvariants. Therefore we must add properties to check that any changeSchema's follow the state restrictions. Part of the birthdaybook Gpsa is shown in figure ???. 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
stateInvariants SI1
initialSchema IS1
postcondition P02
outputSchema OS1
precondition PRE2
changeSchema CS1
totaliseSchema TS1
.....
lemma L1_(CS1)
```

Figure 6.17: Part of the GPSa for the birthdayBook example. (Full version shown in appendix ??)

6.5 Conclusion

This chapter describes how the ZDRa program uses the GoTo graph to generate a proof sketch (step 2→3 in figure ??). The proof sketch 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 consistency of the state invariants and show an example. The next chapter describes how the proof sketch is translated into Isabelle syntax.

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

L_1, L_2, L_3

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 proof sketch;
22  | end
23 else
24  | nothing
25 end
```

Algorithm 3: Part of the algorithm to create Proof Obligation ZDRa names.

Chapter 7

General Proof Sketch aspect and beyond

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 (without a context) and named it ‘a’ then the program will create an empty Isabelle skeleton e.g.

```
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 specification named ‘a’ we get the following (after the preivable described before):

```
record SS1 =
(*DECLARATIONS*)
locale a =
fixes (*GLOBAL DECLARATIONS*)
assumes SI#
```

```
begin
end
end
```

If there are no state invariants 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 written in ZDRa:

```
\draschema{CS1}{

\begin{schema}{b}
someDeclaration

\where
\dra{P01}{someExpression}

\end{schema}}
```

Then when translating into the Isabelle skeleton it becomes:

```
definition CS1 ::

"(*CS1_TYPES*) => bool"

where

"CS1 (*CS1_VARIABLES*) == P01"
```

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:

```
\dra{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 GPSa into Isabelle syntax depends if the changeSchema in question has a precondition, postcondition or both. We use definition ?? 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

```

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.

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 i.e. 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.

7.2 ZCGa specification to Fill in the Isabelle Skeleton

Since translating using ZMathLang can even been done on incomplete specifications, it is important to note that if some 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.

7.2.1 Types and Freetypes in Z

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 `typedef 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.

7.2.2 Declarations

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

```
\draschema{OS1}{

\begin{schema}{ab}

d: \power COLOUR

c: COLOUR

\where

\draline{P01}{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*) == (P01)"
```

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*)`.

7.2.3 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 ‘P01’ which is then replaced by the expression $c \backslash<\text{in}> d$. Note this expression is very similar to the expression in L^AT_EX $c \backslash\text{in } d$ apart from the symbol $\backslash\text{in}$ becomes $\backslash<\text{in}>$. Table ?? shows the rest of these automatic changes of the syntax made from L^AT_EX to Isabelle.

Note: The following table shows one example of mapping Z specifications into the Isabelle theorem prover. There may be other ways to map the syntax from L^AT_EX into Isabelle but we use the following table to show the proof of concept.

Syntax in Z	Syntax in L ^A T _E X	Syntax in Isabelle
{...}	$\backslash\{\dots\}$	{...}
(...)	$\backslash\text{limg}\dots\backslash\text{rimg}$	“ ”
$\langle\dots\rangle$	$\backslash\text{langle}\dots\backslash\text{rangle}$	$\langle\text{langle}\rangle\dots\langle\text{rangle}\rangle$
# A	$\backslash\#$	card if A is set, length if A is list
\cup	$\backslash\text{cup}$	$\langle\text{union}\rangle$
\cap	$\backslash\text{cap}$	$\langle\text{inter}\rangle$
\times	$\backslash\text{cross}$	$\langle\text{times}\rangle$
\	$\backslash\text{setminus}$	-
\geq	$\backslash\text{geq}$	$\langle\text{ge}\rangle$
\leq	$\backslash\text{leq}$	$\langle\text{le}\rangle$
\triangleleft	$\backslash\text{lhd}$	$\langle\text{lhd}\rangle$
\triangleright	$\backslash\text{rhd}$	$\langle\text{rhd}\rangle$
$\triangleright\triangleright$	$\backslash\text{nrres}$	$\langle\text{unlhd}\rangle$
$\triangleleft\triangleleft$	$\backslash\text{ndres}$	$\langle\text{unrhd}\rangle$
\Rightarrow	$\backslash\text{implies}$	$\langle\text{Longrightarrow}\rangle$
\Leftrightarrow	$\backslash\text{iff}$	$\langle\text{Longleftrightarrow}\rangle$
\notin	$\backslash\text{notin}$	$\langle\text{notin}\rangle$
\in	$\backslash\text{in}$	$\langle\text{in}\rangle$
\subset	$\backslash\text{subset}$	$\langle\text{subset}\rangle$
\subseteq	$\backslash\text{subsext}$	$\langle\text{subsext}\rangle$
\wedge	$\backslash\text{land}$	$\langle\text{and}\rangle$
\vee	$\backslash\text{lor}$	$\langle\text{or}\rangle$
\neg	$\backslash\text{lnot}$	$\langle\text{not}\rangle$
\neq	$\backslash\text{neq}$	$\langle\text{noteq}\rangle$
$a \mapsto b$	$a \backslash\text{mapsto } b$	(a,b) ‘ ’ if set preceding using $\langle\text{rightharpoonup}\rangle$
$\mathbb{P}A$	$\backslash\text{power } A$	A set
\mathbb{N}	$\backslash\text{nat}$	nat
\mathbb{N}_1	$\backslash\text{nat_1}$	nat
\mathbb{Z}	$\backslash\text{num}$	num
$A \rightarrow B$	$A \backslash\text{pfun } B$	$(A \langle\text{rightharpoonup}\rangle B)$
$A \rightarrowtail B$	$A \backslash\text{fun } B$	$(A * B) \text{ set}$
$A \leftrightarrow B$	$A \backslash\text{rel } B$	$(A * B) \text{ set}$
$\text{seq } A$	$\backslash\text{seq } A$	A list
$\text{iseq } A$	$\backslash\text{iseq } A$	A list
$\text{seq}_1 A$	$\backslash\text{iseq_1 } A$	A list
$\text{dom } A$	$\backslash\text{dom } A$	Domain A dom if set preceding using $\langle\text{rightharpoonup}\rangle$
$\text{ran } A$	$\backslash\text{ran } A$	Range A ran if set preceding using $\langle\text{rightharpoonup}\rangle$
\exists	$\backslash\text{exists}$	$\langle\text{exists}\rangle$

\forall	<code>\forall</code>	<code>\<forall></code>
\bullet	<code>@</code>	<code>.</code>
R^\sim	<code>R\inv</code>	<code>\<R^-1></code>
R^k	<code>R^{k}</code>	<code>\<R^k></code>

Table 7.1: A table showing the symbols which are changed from Z specifications in L^AT_EX 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 occasions.

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 therefore we translate total functions as a set of pairs `(A * B) set`, we use the `\<rightharpoonup>` symbol to describe partial functions in the isabelle syntax.

In the following section we highlight the symbols in the Z toolkit in red and the corresponding Isabelle translation in blue.

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 the ‘\mapsto’ term is in a total function where the Z syntax is ‘A \fun B’ then the ‘\mapsto’ term will be translated as ‘(f,s)’ in isabelle. However if the ‘\mapsto’ term is in partial function set then the ‘\mapsto’ term will be translated as ‘f s’. For example if ‘`funset: A \fun B`’ and ‘`c \mapsto d \in funset`’ then the isabelle translation will be $(c,d) \in funset$ however if ‘`pfunset: A \pfun B`’ and ‘`c \mapsto d \in pfunset`’ then the isabelle translation will be $c \ d \in pfunset$.
- ‘\dom’ and ‘\ran’ in Z syntax are the same regardless whether it is being applied to an element in a partial function or a total function. However in

¹by list we mean an ordered set

Isabelle the syntax varies between the two types of functions. If the ‘\dom’ being applied to an element ‘k’, with type ‘A \fun B’ then the Isabelle translation would be ‘Domain k’ whereas if ‘k’ has type ‘A \pfun B’ then it will be translated as ‘dom k’. Similarly we would have ‘Range k’ and ‘ran k’ for total and partial functions of ‘k’ respectively. For example if ‘funset: A \fun B’ and ‘\dom k \in funset’ then the isabelle translation will be $(A * B) set$ and $Domain k \in funset$. however if ‘pfunset: A \pfun B’ and ‘\dom k \in pfunset’ then the isabelle translation will be $A \rightarrow B$ and $dom k$.

7.2.4 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}{..}} then anywhere ‘TS1’ is listed in the skeleton it will be converted to ‘ab’. This is done throughout the entire skeleton.

7.2.5 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{P03}{\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}{P03}
```

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

```
definition AddBirthday ::

  "(NAME set) \<Rightarrow> NAME \<Rightarrow> BirthdayBook
  \<Rightarrow> BirthdayBook \<Rightarrow> DATE \<Rightarrow>
  (NAME \<rightarpoonup> DATE) => bool"

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 is represented as the following in the Isabelle Skeleton:

```
lemma CS1_L1:

  "(\<exists> (*CS1_VARIABLESANDTYPES*).
  (PRE1)
  \<and> (P03)
  \<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'))"
```

7.3 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 ??, 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 [?] may be useful when proving the properties.

Figure ?? 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 we have used sledgehammer [?] which is part of the Isabelle/Hol package. The full proof of this specification can be found in appendix ???. At this point, proving properties in a theorem prover may require some expertise in the field. Proving tools such as SMT solvers [?] and sledgehammer are an interesting area of research on its 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 \\<subseteqq>
  students) \\<and> (Range taking \\<subseteqq> degModules) \\<and> (Domain taking'
  \\<subseteqq> students') \\<and> (Range taking' \\<subseteqq> degModules')))"
by blast
```

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

7.4 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 annotated specification giving a halfbaked proof. The last step would be to fill in any missing proofs in the halfbaked proof (as show in the proof obligation in figure ??), 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 [?] may be useful at this point. We will demonstrate how to go from a Z specification to a proven specification in Isabelle in chapter ??.

Chapter 8

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 than by just typing commands in a terminal. Full details for the user interface and all its functions can be found in Mihaylova's user guide [?]. 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.

8.1 Inserting a specification

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

```
laptop:~/laviniyas_workspace/example$ python Interface.py
```

Figure 8.1: Example of how to start the interface for the ZMathLang framework using the terminal.

An example of the interface can be seen in figure ???. 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 ??.

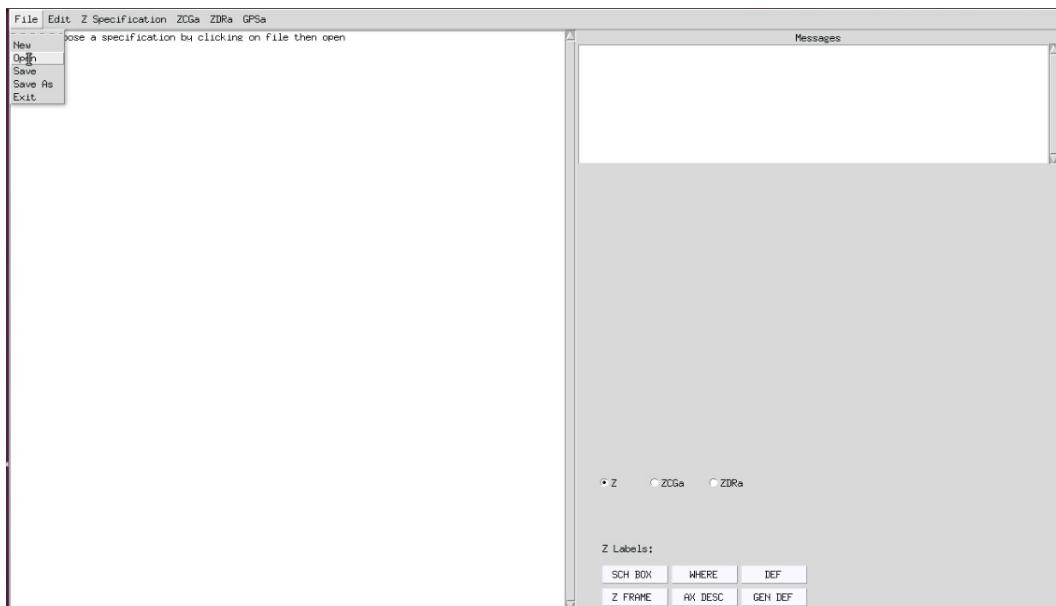


Figure 8.2: This figure shows the steps to open an existing specification to be imported into the user interface.

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 ??.

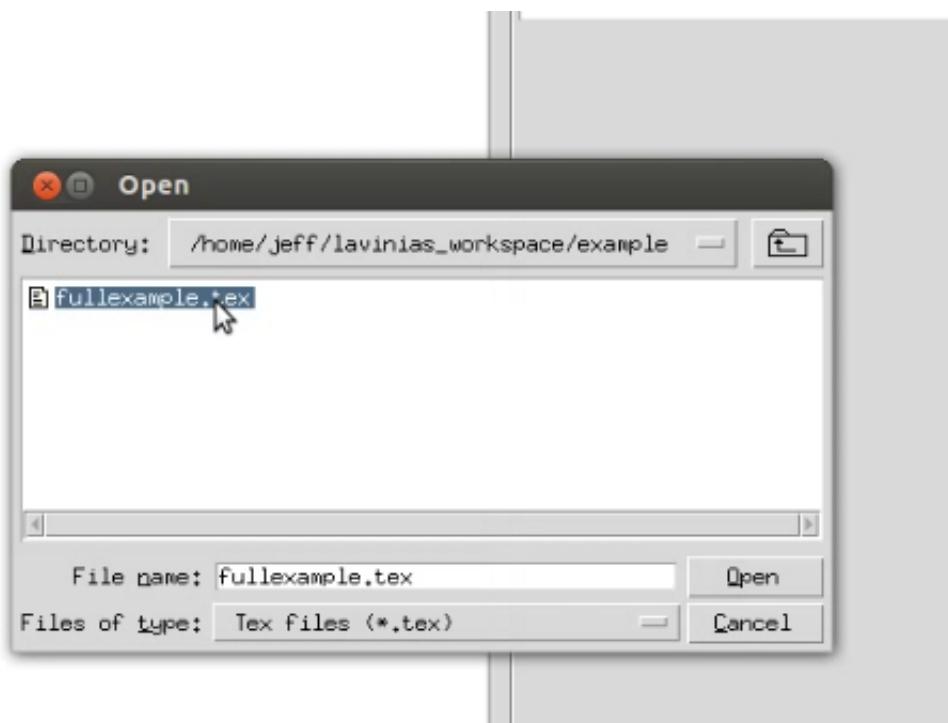


Figure 8.3: This figure shows the pop up window that comes up when a user is asked which specification they wish to insert. In our example we would like to import a file named "fullexample.tex".

Once a specification has been chosen then the specification should appear in the panel on the left hand side. Figure ?? shows an example of this. Note that no messages appear yet in the messages box.

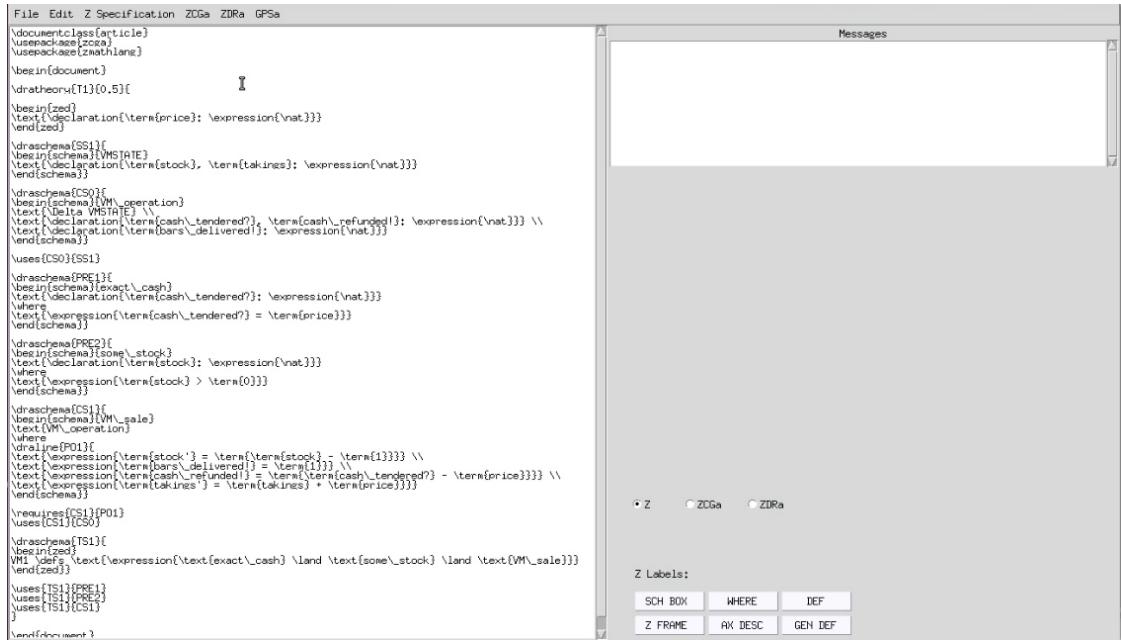


Figure 8.4: This figure shows an imported in the user interface which is shown in the main panel on the left.

8.2 Checking ZCGa

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



Figure 8.5: An example of how to check the specification for ZCGa correctness (left) and the message which appears when the specification is ZCGa correct (right).

To check for ZCGa correctness the user can click on the *zcg*a button from the top menu and then click on *Zcg Check*. If the specification is grammatically correct then a message appears in the message box (see figure ??).

8.3 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).

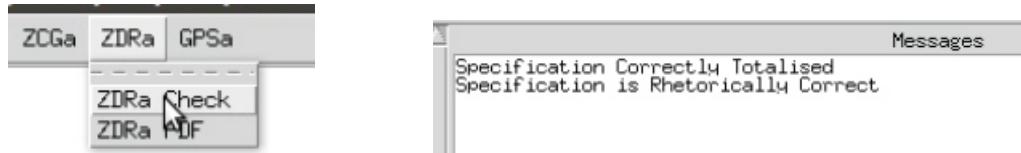


Figure 8.6: An example of how to check the specification for ZDRa correctness (left) and the message which appears when the specification is ZDRa correct.

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 ?? shows both of these actions.

8.4 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.

8.4.1 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* from the drop down menu. An example of this is shown in figure ??.

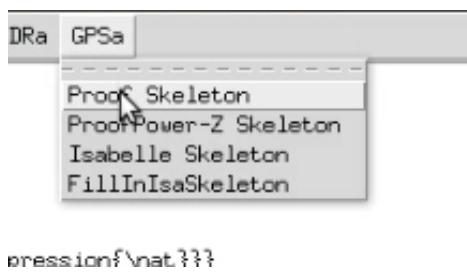


Figure 8.7: This part of the user interface shows the steps to create a general proof skeleton from the users loaded specification..

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



Figure 8.8: A new file appears named skeleton in the same directory as the users specification. This is the automatically generated 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 ??). By clicking on this new button the general proof skeleton can be opened within the Interface.

8.4.2 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 ??.

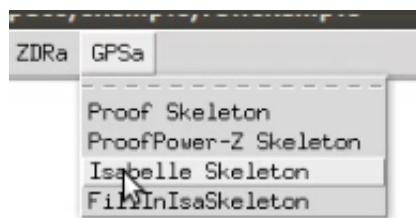


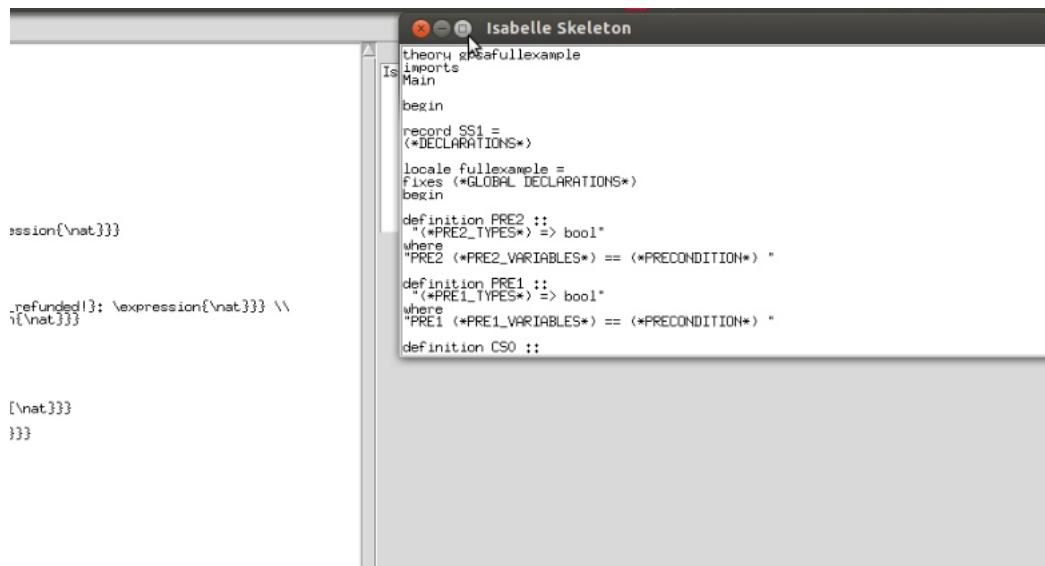
Figure 8.9: To create an Isabelle skeleton then '*Isabelle Skeleton*' should be selected from the GPSa sub menu on the user interface.

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 ???. A new file will be produced and automatically saved in the same directory as the interface with a `.thy` extension.



Figure 8.10: New buttons which appear on the right hand side of the interface 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 clicking 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 ??.



```

theory gosafullexample
imports Main
begin

record SS1 =
(*DECLARATIONS*)

locale fullexample =
fixes (*GLOBAL DECLARATIONS*)
begin

definition PRE2 :: "(*PRE2_TYPES*) => bool"
where
"PRE2 (*PRE2_VARIABLES*) == (*PRECONDITION*)"

definition PRE1 :: "(*PRE1_TYPES*) => bool"
where
"PRE1 (*PRE1_VARIABLES*) == (*PRECONDITION*)"

definition CS0 ::

[(*nat*)]
}

```

Figure 8.11: By clicking ‘Show Isabelle skeleton’ a box will be displayed 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 ??). 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 ??).

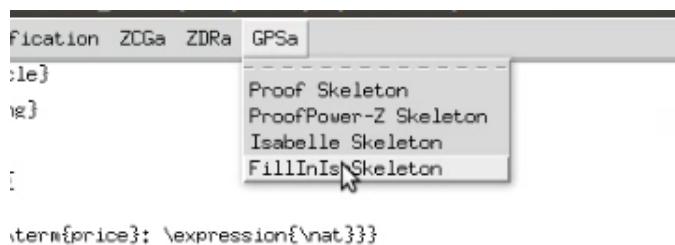


Figure 8.12: This figure shows the steps to automatically fill in the Isabelle skeleton. The user can choose the ‘*FillInIsaSkeleton*’ from the GPSa sub menu.

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.

8.5 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 ?? 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 correctly totalised	Not all preconditions have an alternative output (skeleton can still be created) (<i>see chapter ??</i>)
Specification is Rhetorically Correct	Specification is ZDRA correct (<i>see chapter ??</i>)
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	Convert the specification into an Isabelle skeleton first before filling it in (<i>see chapter ??</i>)
Please select your isabelle skeleton:	Please locate the Isabelle skeleton which you wish to be filled in (<i>see chapter ??</i>)
Please convert specification into GPSa first	Can not find the general proof skeleton (<i>see chapter ??</i>)
Loops in reasoning Can not create Skeleton	Specification is not ZDRA correct and the skeleton can not be created (<i>see chapter ??</i>)
Spec Grammatically Correct	The specification has passed ZCCa checker
Spec Grammatically Incorrect Number of errors: 2	The specification has failed ZCCa correctness and has 2 errors (<i>see chapter ??</i>)

Table 8.1: Messages which could appear in the user interface and their meanings.

The user interface allows the users to create, import and edit a specification. It allows the user to annotate their specification and check for ZCGa correctness. The user may also use the interface to annotate and check for ZDRA correctness. The GPSa, Isabelle skeleton, dependency and GoTo may also be created via the user interface. The only part which currently may not be done via the user interface is the final proofs in Isabelle. Here the user will need to complete the proof of their formal specification via the Isabelle user interface or another program which can parse Isabelle. Future work on ZMathLang may include add on an Isabelle plug-in to the ZMathLang user interface. The user will only need to use the terminal to

start the ZMathLang user interface. All other steps are done via the ZMathLang or Isabelle interface (in the case of the final step).

8.6 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. The interface also allows the user to view the documents automatically produced from the annotated specification.

Chapter 9

From raw specification to fully proven spec: A full example

In this chapter we take two system specifications 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.

9.1 ModuleReg Example

We have chosen to demonstrate the Modulereg specification from start to finish using the steps of ZMathLang as the modulereg specification contains 2 state invariants which means we can demonstrate ZMathLang automatically generating the lemmas to prove. The Modulereg specification is also a good size to fit onto one page to make it easy for the reader.

9.1.1 Step 0

Raw Specification

We take the raw specification ModuleReg [?], which displays students taking modules in a school environment (shown in figure ??) The output for the specification

can be seen in figure ???. The full proof for the ModuleReg specification can be found in ??.

```
\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 9.1: Part of the raw schema.

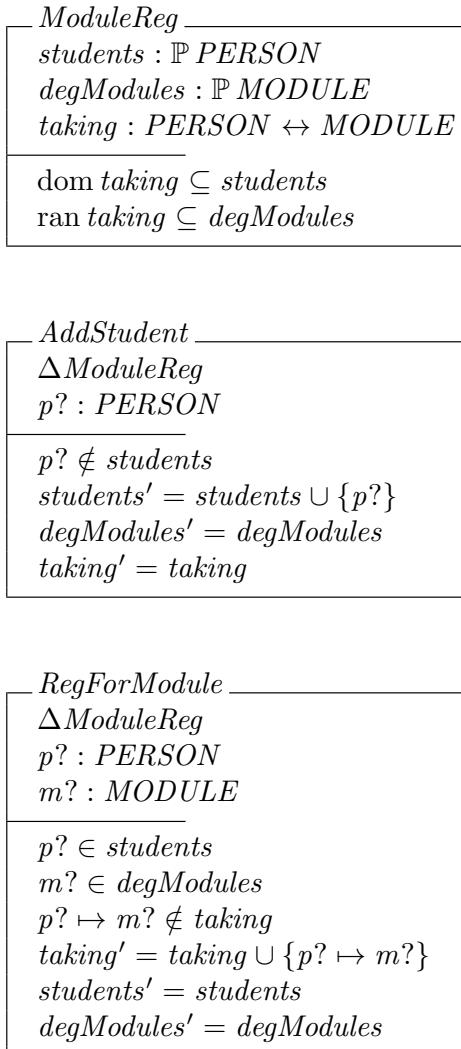


Figure 9.2: Part of a raw specification output.

9.1.2 Step 1

ZCGa

The user then goes to label the raw specification with the ZCGa labels. The labelled specification can be seen in figure ???. The words highlighted in red are the ZCGa annotations done by the user and the black text is the existing specification. The output from figure ?? is shown in figure ??.

```
\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}}}} \\
\subsetseteq \set{students}\\
\text{\expression{\set{\ran \set{taking}}}} \\
\subsetseteq \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{students} \cup \set{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{p?} \mapsto \term{m?} \notin \set{taking}}}}\\
\text{\expression{\set{taking'} = \set{taking} \cup \set{\term{p?} \mapsto \term{m?}}}}}}\\
\text{\expression{\set{students'} = \set{students}}}}\\
\text{\expression{\set{degModules'} = \set{degModules}}}}
\end{schema}
\end{document}
```

Figure 9.3: Part of the raw schema.

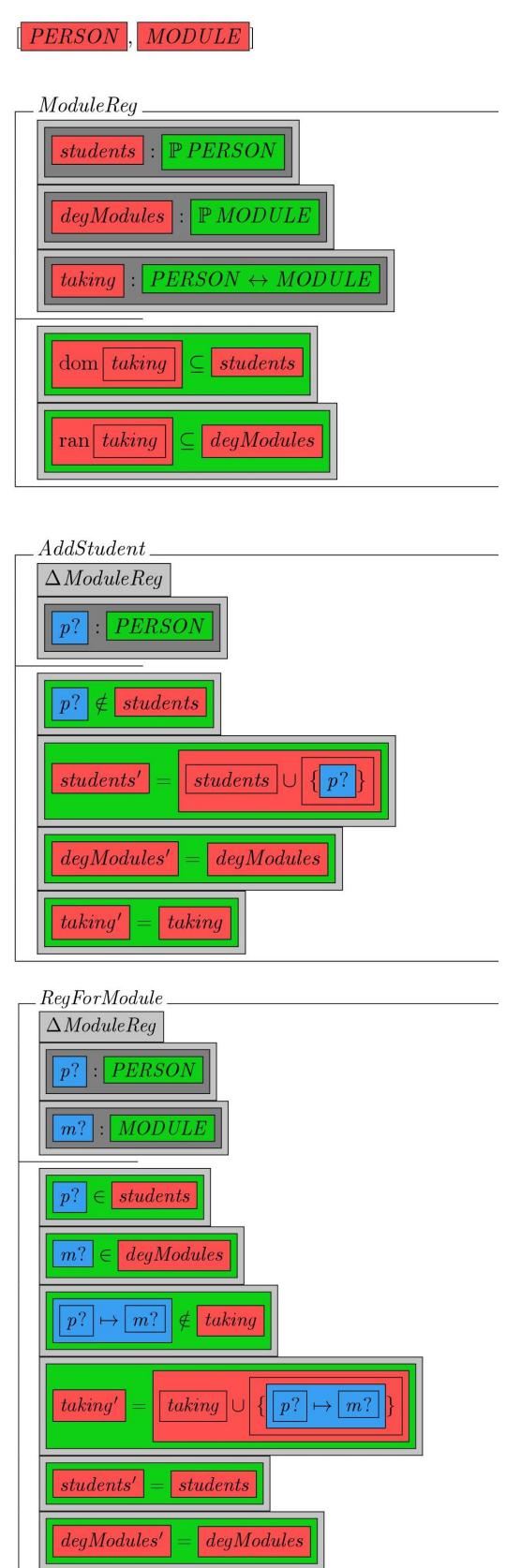


Figure 9.4: Part of a ZCGa labelled specification output.

After annotating the text, we run it through the ZCGa correctness checker.

Figure ?? shows the message which appears when the annotated specification has been checked.

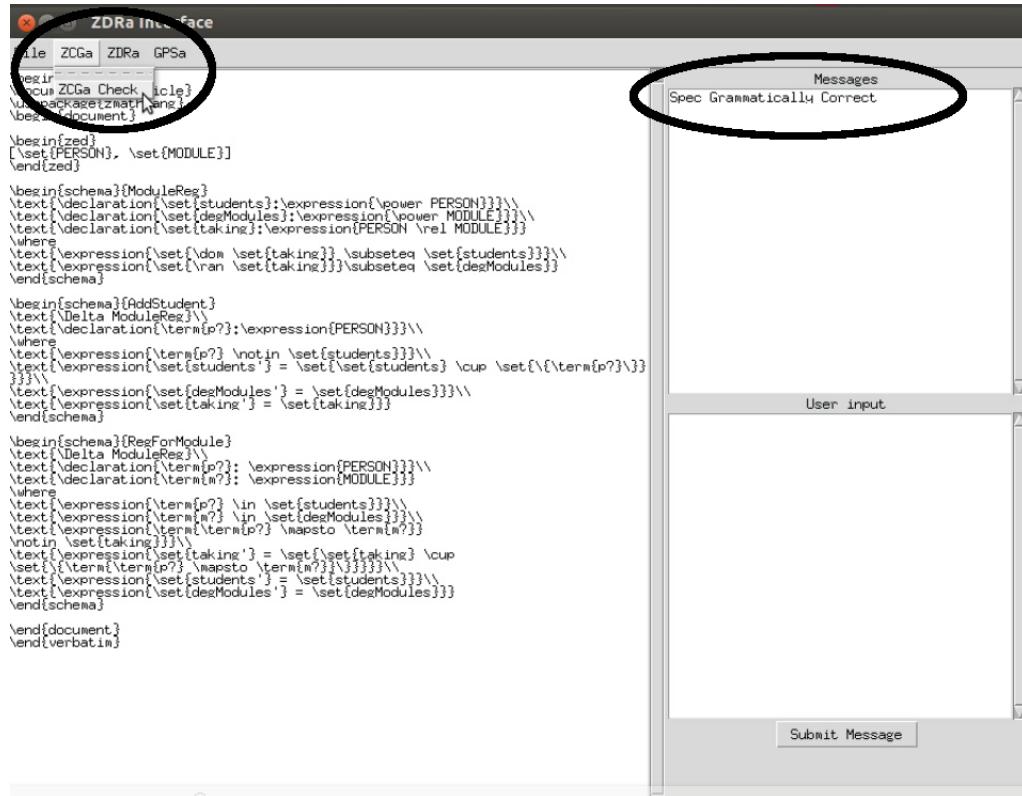


Figure 9.5: Message which appears after running the ZCGa checker on our example.

9.1.3 Step 2

ZDRA

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

```

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

\dratetheory{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}}}
\end{schema}

\where

\draline{SI1}{ \text{\expression{\set{\dom{taking}} \subset \set{students}}}}\\
\text{\expression{\set{\ran{taking}} \subset \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{\notin{\set{students}}{p?}}}}\\
\draline{PO1}{ \text{\expression{\set{students} = \cup \set{\term{p?}}}}}}\\
\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}}:\\ \expression{MODULE}}}
\end{schema}
}

```

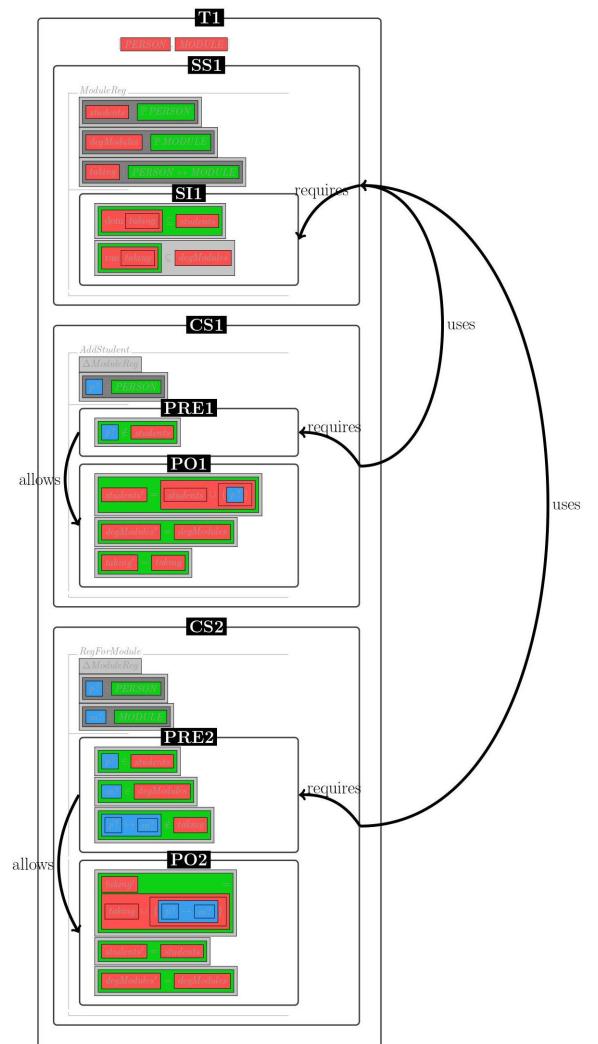


Figure 9.6: An example of a specification labelled in ZCGa and ZDRa.

Figure 9.7: 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 ?? shows the message which appears after we check our example with ZDRa.

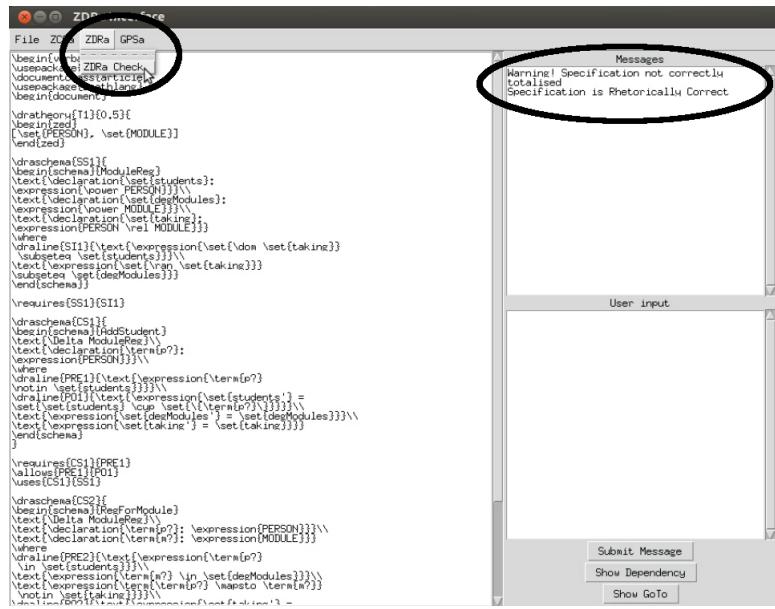


Figure 9.8: Message which appears after running the ZDRa checker on our example.

9.1.4 Step 2.5

Graphs

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

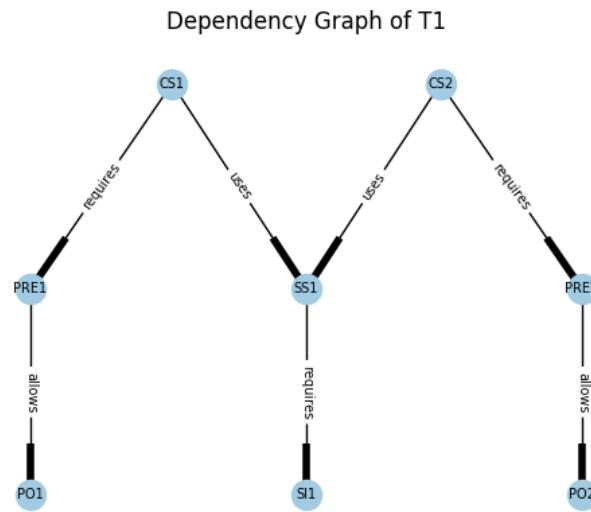


Figure 9.9: Dependency graph automatically generated from the ZDRa for our example.

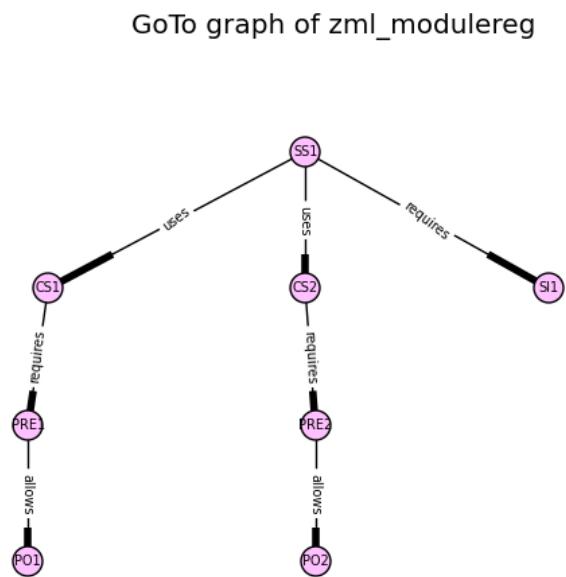


Figure 9.10: GoTo graph automatically generated from the ZDRa for our example.

9.1.5 Skeletons

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

9.1.5.1 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 ??).

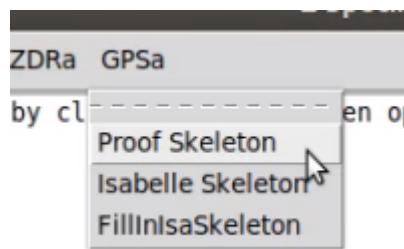


Figure 9.11: The GPSa button allows the user to generate the general proof skeleton.

The GPSa can be created after a specification has been annotated with ZDRa. The GPSa is what orders the specification logically to put into a theorem prover. Steps 4, 5 and 6 will need to be created after the GPSa. Step 4 will use the skeleton created in step 3 and ZCGa annotations created in step 2 to create the Isabelle skeleton. The Isabelle skeleton in step 4 requires ZCGa annotations and the GPSa to be complete. Without the GPSa, the logical order of the specification may not be correct e.g. there may be some changeSchemas used which haven't been created yet. Without the ZCGa, some variables may be used which haven't been declared and therefore will not parse through Isabelle.

Figure ?? 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 there 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
stateInvariants SI1
changeSchema CS2
precondition PRE2
changeSchema CS1
precondition PRE1
postcondition PO2
postcondition PO1
lemma L1_(CS2)
lemmaL2_(CS1)
```

Figure 9.12: General proof skeleton.

9.1.5.2 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 ??.

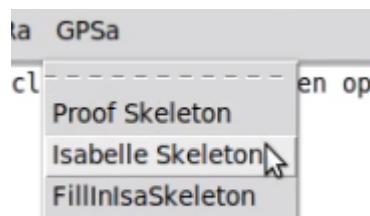


Figure 9.13: The Isabelle skeleton button allows the user to generate an Isabelle skeleton of their specification.

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 in between (* ... *) parenthesis to show the parts which need to be filled in either by using the ZCGa document or by the user. Figure ?? shows the automatically generated Isabelle skeleton for our `modulereg` example.

```

theory gpsaModuleReg
imports Main begin (*DATATYPES*)
record SS1 = (*DECLARATIONS*)
locale lnt = fixes (*GLOBAL DECLARATIONS*)
assumes SI1 begin definition CS2 :: 
"(*CS2_TYPES*) => bool"
where
"CS2 (*CS2_VARIABLES*) == (PRE2) \<and> (P02)"
definition CS1 :: "(*CS1_TYPES*) => bool"
where
"CS1 (*CS1_VARIABLES*) == (PRE1) \<and> (P01)"

lemma CS2_L1: "(\<exists> (*CS2_VARIABLESANDTYPES*).
(PRE2) \<and> (P02) \<longrightarrow>
(SI1) \<and> (SI1'))"
sorry
lemma CS1_L2: "(\<exists> (*CS1_VARIABLESANDTYPES*).
(PRE1) \<and> (P01) \<longrightarrow>
((SI1) \<and> (SI1')))" sorry
end
end

```

Figure 9.14: Isabelle proof skeleton.

9.1.5.3 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 interface and then click on ‘FillInIsa Skeleton’ in the sub menu (shown in figure ??).

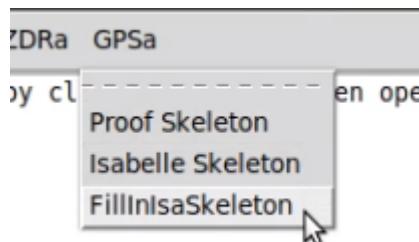


Figure 9.15: The ”FillInIsa Skeleton” button allows the user to fill in the skeleton they previously created.

Figure ?? shows our example with a filled in Isabelle skeleton. It is important to note that the program also changes some of the syntax from L^AT_EX to Isabelle so that it is fully parsable by Isabelle. We have used the rules described in section ?? to fill in the Isabelle skeleton with the ZCGa annotations.

```

theory gpsaModuleReg imports Main

begin

  typecdecl PERSON typecdecl 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 \<subseteqq> students" and
  "Range taking \<subseteqq> 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) \& (m
      \<in> degModules) \& ((p, m) \<notin> taking)
      \& (taking' = taking \<union> {(p, m)}) \&
      (students' = students)
      \& (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)
      \& (students' =
      students \<union> {(p)}))

      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)
      \& (m \<in> degModules) \& ((p, m)
      \<notin> taking) \& (taking' =
      taking \<union> {(p, m)})) \&
      (students' = students)
      \& (degModules' =
      degModules)) \<longrightarrow> ((Domain taking
      \<subseteqq> students) \&
      (Range taking \<subseteqq> degModules) \&
      (Domain taking' \<subseteqq>
      students') \&
      (Range taking' \<subseteqq> 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)})) \&
      (degModules' = degModules) \&
      (taking' = taking))
      \<longrightarrow> ((Domain
      taking \<subseteqq> students) \&
      (Range taking \<subseteqq> degModules) \&
      (Domain taking' \<subseteqq> students')) \&
      (Range taking' \<subseteqq> degModules')))" sorry end end
    
```

Figure 9.16: Filled In proof skeleton.

Figure ?? 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 consistency 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, ZMathLang is unable to prove the lemma's automatically and it is up to the user to do this (explained 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.

In this case the new ”Lemmas” are added to the end of the specification at random. For this specification it doesn’t matter which lemma comes first (`RegForModule_L1` or `AddStudent_L2`) as the schemas `RegForModule` and `AddStudent` are independent of each other and one does not ‘*use*’ the other. If we did have a `changeSchema` which did use another `changeSchema` then this would have to be annotated in the ZDRA and therefore the order would matter when the lemmas are produced.

9.1.6 Step 6

Full Proof

The next part is to prove any existing lemmas from the filled in Isabelle Skeleton 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 [?] to help them prove the properties. An example is shown in figure ?? 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) \& (m \<in> degModules))
\<and> ((p, m) \<notin> taking) \<and>
(taking' = taking \<union> {(p, m)})
\<and> (students' = students) \<and>
(degModules' = degModules))
\<longrightarrow> ((Domain taking \<subseteqq> students) \<and> (Range taking \<subseteqq> degModules)) \<and>
(Range taking' \<subseteqq> 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 \<subseteqq> students) \<and>
(Range taking \<subseteqq> degModules)) \<and>
(Domain taking' \<subseteqq> students') \<and>
(Range taking' \<subseteqq> degModules'))"
by blast

```

Figure 9.17: An example of a property and its proof for the Module Reg example.

Figure ?? shows how the lemmas are 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. The user has completed this proof by using techniques written in Isabelle, there are some forms of help by automatically solving proofs using the ”sledgehammer” and ”smt” provers which is what the user used to help solve these proofs. Proofs can be completed in a variety of different ways and there are so many different strategies to complete a single proof. Automatically solving proofs can be a whole research area in itself. More information on proving these lemmas can be found in the Isabelle Manual [?] and is beyond the scope of this thesis.

We know that the Isabelle skeleton is now correct as it can be parsed through the Isabelle theorem prover with no errors. All variables have been declared before they are used in Isabelle and all the ZCGa names have been translated back to the

original names used in the Z Specification.

9.2 Autopilot Specification

Here, we show a section of an autopilot specification [?] in natural language, which describes the control of an aircraft without constant manual input by the human pilot. We have chosen to demonstrate this specification as it presents the ZMathLang toolkit on a specification which is semi-formal.

9.2.1 Step 0

Raw Specification

```

\documentclass{article}
\begin{document}
\begin{enumerate}
\item 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:
\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}
\end{enumerate}

\begin{zed}
events ::= press\_att\_cws | press\_cas\_eng |
press\_alt\_eng | \\
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 alt\_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}
mode\_status ::= off | engaged
\end{zed}

\begin{schema}{off\_eng}
mode: mode\_status
\where
mode = off \lor mode = engaged
\end{schema}

\begin{schema}{AutoPilot}
att\_cws: mode\_status ||
fpa\_sel: mode\_status ||
alt\_eng: mode\_status ||
cas\_eng: mode\_status
\end{schema}
\begin{schema}{att\_cwsDo}
\Delta AutoPilot
\where

```

`att_cws = off \\`
`att_cws' = engaged \\`
`fpa_sel' = off \\`
`alt_eng' = off \\`
`cas_eng' = off \lor engaged \\`
`\end{schema}`

`\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 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 alt_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 alt_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.`

`\item 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.`

Figure 9.18: Part of the raw specification of the Autopilot example

When we compile the raw L^AT_EX specification shown in figure ?? we get the output in ???. This output is very similar to the semi-formal specification written in figure ??.

??.

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 | press_cas_eng | press_alt_eng |
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 alt_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 | engaged
```

off_eng
mode : mode_status
mode = off \vee mode = engaged

AutoPilot
att_cws : mode_status
fpa_sel : mode_status
alt_eng : mode_status
cas_eng : mode_status

att_cwsDo
Δ AutoPilot
att_cws = off
att_cws' = engaged
fpa_sel' = off
alt_eng' = off
cas_eng' = off \vee 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 alt_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 alt_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 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 modeled.

Figure 9.19: Part of the raw specification of the Autopilot example

9.3 Step 1

ZCGa

The user can then start labelling the semi-formal specification using the ZCGa annotations. As this specification is not complete, the ZCGa labelling is not complete either. However, we can already see the benefits the ZCGa weak type checker can bring to an semi-formal specification.

```

\documentclass{article}
\usepackage{zmathlang}
\begin{document}
\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}
\end{enumerate}
\begin{set}{events} ::= \term{press\_att\_cws} | \term{press\_cas\_eng} | \term{press\_alt\_eng} | \term{press\_fpa\_sel}
\begin{zed}
Only one of the first three modes can be engaged at any time. However, the can be engaged at the same time as any of the other modes. The pilot engages the corresponding button on the panel. One of the three modes, att\_cws, f should be engaged at all times. Engaging any of the first three modes will other two to be disengaged since only one of these three modes can be enga
\begin{zed}
\set{mode\_status} ::= \term{off} | \term{engaged}
\end{zed}
\begin{schema}{off\_eng}
\text{\declaration{\term{mode}: \expression{mode\_status}}}
\where
\text{\expression{\expression{\term{mode} = \term{off}}} \lor \expression{\term{mode} = \term{engaged}}}
\end{schema}
\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}
\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}
\begin{schema}{att\_cwsDo}
\text{\Delta{AutoPilot}}
\text{\term{att\_cws} = off} \\
\text{\term{att\_cws'} = engaged} \\
\text{\term{fpa\_sel'} = off} \\
\text{\term{alt\_eng'} = off} \\
\text{\term{cas\_eng'} = off} \vee \text{\term{cas\_eng'} = engaged}
\end{schema}
\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 path angle, and air speed of the aircraft. However, the pilot can enter a dialling in the value using the knob next to the display. This is the target the pilot wishes the aircraft to attain. For example, if the pilot wishes 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.

```

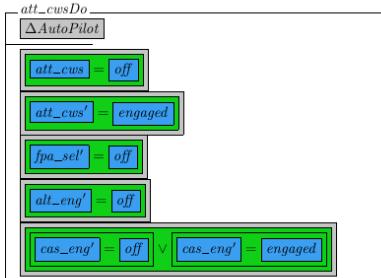
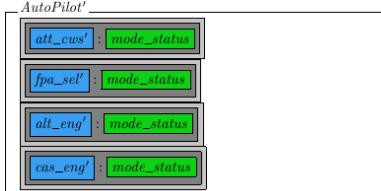
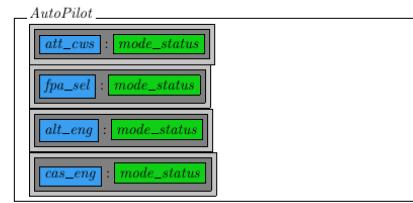
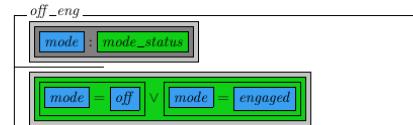
1. The mode-control panel contains four buttons for selecting modes and three displays for dialling in or displaying values. The system supports the following four modes:

- 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 | press_cas_eng | press_alt_eng | 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 alt_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 | 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 dialling in the value using the knob next to the display. This is the target or "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

Figure 9.20: Part of the autopilot specification labeled in ZCGa.

Figure ?? shows part of the semi-formal autopilot specification labelled in ZCGa.

At the moment this specification is ZCGa correct. One of the main benefits of the ZCGa is that the checker only checks what is annotated, therefore the user may wish to run this checker whilst the specification is not complete.

1. 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:

- 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 `alt_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.

Messages
 ['cas_eng', 'att_cws',
 'fpa_sel', 'alt_eng'] types
 not declared
 Spec Grammatically
 Incorrect
 Number of errors 4

Figure 9.21: Part of the autopilot specification which is ZCGa incorrect.

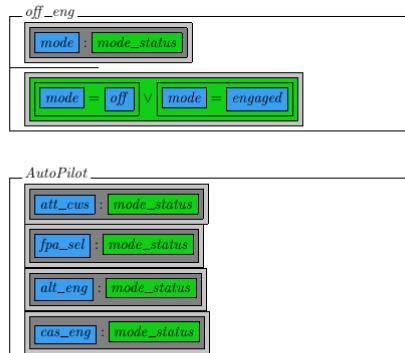
If the user annotated terms within a paragraph without declaring them (as shown in figure ??) then the ZCGa would identify this as being incorrect.

The user would need to declare his terms first before using them within the paragraph. This is shown in figure ??.

1. 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:

- attitude control wheel steering (att_cws)
- flight path angle selected (fpa_sel)
- altitude engage (alt_eng)
- calibrated air speed (cas_eng)

`mode_status` ::= `off` | `engaged`



Messages
 Spec Grammatically Correct

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 `alt_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.

Figure 9.22: Part of the autopilot specification which is ZCGa correct.

9.4 Step 2

ZDRA

The user can then add ZDRA annotations to chunk part of the specification together and add relations together. These chunks can be from the informal or formal part of the specification. Figure ?? shows the Autopilot specification labelled in ZDRA (blue) and ZCGa (grey) and figure ?? shows the compiled result.

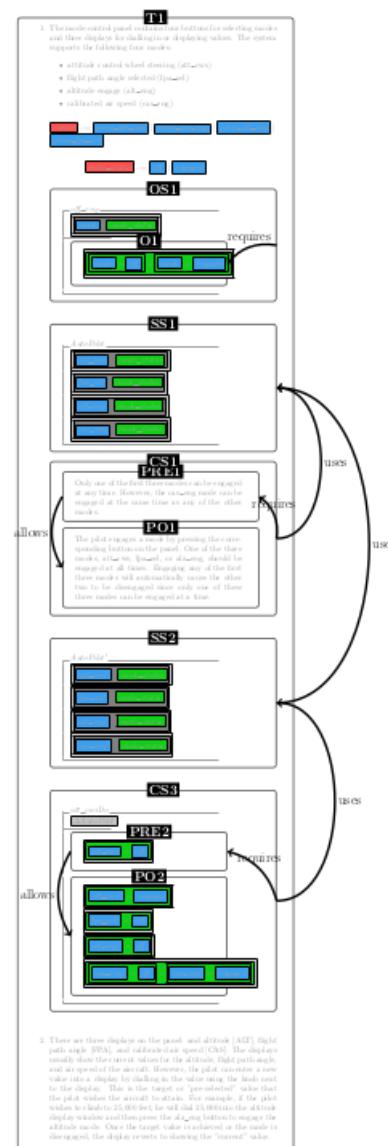


Figure 9.23: Compiled results of the autopilot specification written in ZCGa and ZDRA

```
\documentclass{article}
\usepackage{zmathlang}
\begin{document}
\drattheory{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}
\end{enumerate}

\begin{zed}
\set{events} ::= \term{press\_\att\_\cws} | \term{press\_\cas\_\eng} | \term{press\_\alt\_\eng} | \term{press\_\fpa\_\sel}
\end{zed}

\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
\draline{01}{

\text{\expression{\term{mode} = \term{off}} \lor
\expression{\term{mode} = \term{engaged}}}
\end{schema}
\requires{OS1}{01}

\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{CS1}{\draschema{PRE1}{

Only one of the first three modes can be engaged at any time. However, the \term{cas\_eng} mode can be engaged at the same time as any of the other modes.}

\draschema{P01}{The pilot engages a mode by pressing the corresponding button on the panel. One of the three modes, \term{att\_\cws}, \term{fpa\_\sel}, or \term{alt\_\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.}

\requires{CS1}{PRE1}\allows{PRE1}{P01}\uses{CS1}{SS1}

\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}

\end{zed}

\uses{SS2}{SS1}
\draschema{CS3}{

\begin{schema}{att\_\cwsDo}
\text{\Delta AutoPilot}
\where
\draline{PRE2}{

\text{\expression{\term{att\_\cws} = \term{off}}} \\
\draline{P02}{

\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 \\
\text{\expression{\term{cas\_\eng} = \term{engaged}}}}
\end{schema}
\end{zed}

\uses{CS3}{SS2}\allows{PRE2}{P02}\requires{CS3}{PRE2}

\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.

```

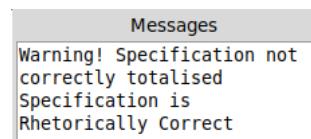


Figure 9.24: The autopilot specification written in ZCGa and ZDRA and the message displayed when run through the ZDRA checker.

Since the semiformal specification is ZDRA correct the dependency and goto graph are automatically produced. These are shown in figure ??.

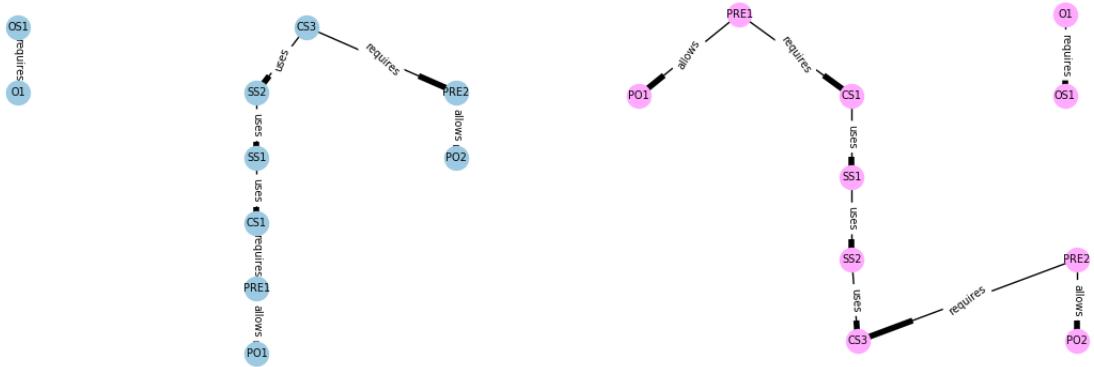


Figure 9.25: Dependency graph (left) and goto graph (right) of the semiformal autopilot specification

9.4.1 Informal Specification

One of the benefits of the ZDRA check is that the chunks of text can even be written informally an example of this is shown in figure ??.

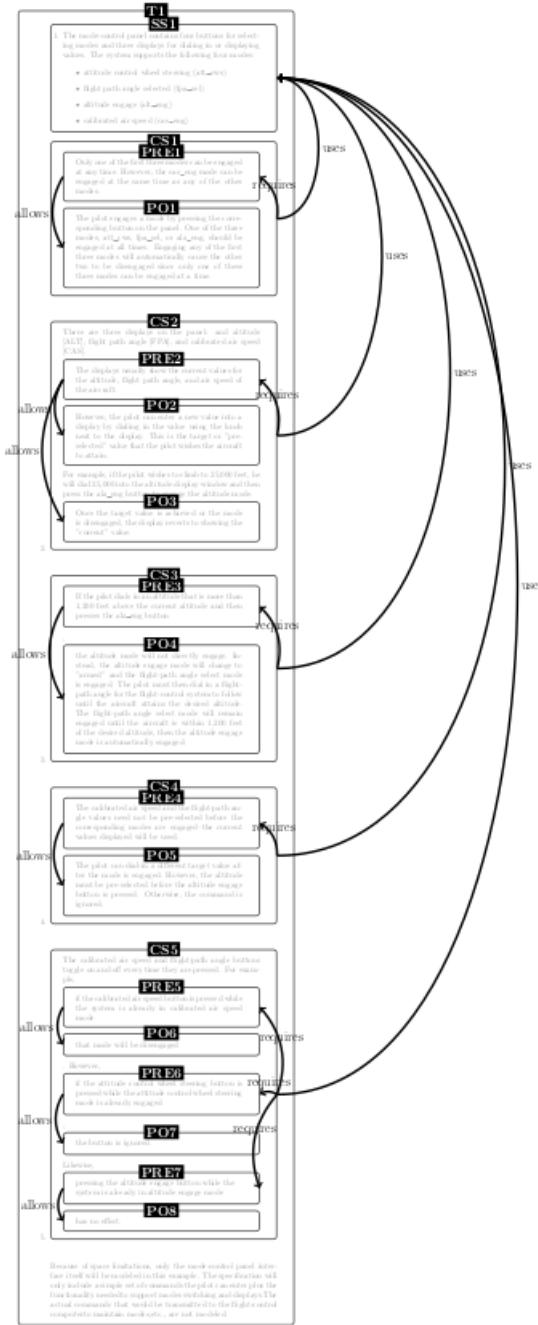


Figure 9.26: Informal specification labelled in ZDRa

If we run the ZDRa labelled specification shown in figure ?? we get the message that it is ZDRa correct however it has not been totalised. Which in this case is legal as it is an informal specification. Future to the message the dependency and goto graphs are also generated. This is show in figure ???. This is different to the dependency and goto graphs of the semi-formal autopilot as the semi-formal specification has formal parts and therefore more details have been added to the specification.

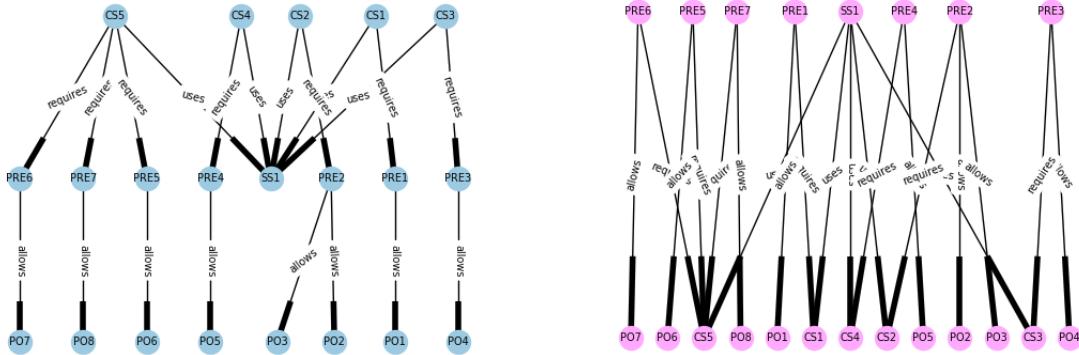


Figure 9.27: Dependency graph (left) and goto graph (right) of the semiformal autopilot informal specification

If the specification had a loop in the reasoning. For example if the description of the mode-control panel was written in SS1 and in CS1 then this would be classed as a loop in the reasoning. We have highlighted in yellow where the description is duplicated and therefore would fail the ZDRA type check.

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

\begin{document}

\drathtory{T1}{0.4}{

\begin{enumerate}

\draschema{SS1}{\item 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:
\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}}

```

Figure 9.28: Loop in the informal reasoning of the autopilot specification (highlighted in yellow)

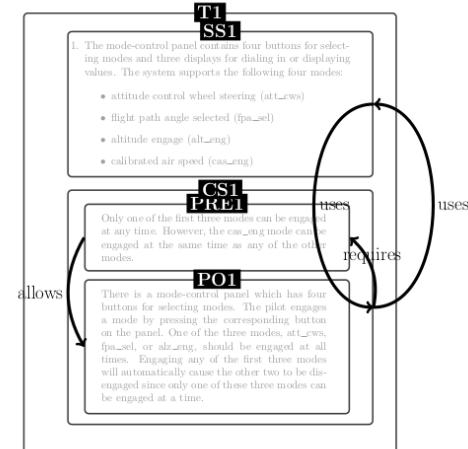


Figure 9.29: A loop in the reasoning of an informal specification

9.4.2 Skeletons

Since our semiformal specification pass the ZDRa check the dependency and goto graphs are automatically generated. The skeletons which could be automatically generated for the semiformal specification are the general proof skeleton (step 3) and the Isabelle skeleton (step 4). ZMathLang can also fill in the Isabelle skeleton for those parts which are in formal format and correctly labelled in ZCGa. The user will not be able to complete a full proof of the specification (step 6) as the semi-formal specification isn't complete as it is not fully formal.

9.4.3 Step 3

General Proof Skeleton

Using the goto graph produced for the semiformal specification in figure ?? the ZMathLang tool-set can automatically generate a general proof skeleton to order the instances for them to be translated into Isabelle. This general proof skeleton is shown in figure ??.

```
stateSchema SS1
precondition PRE1
changeSchema CS1
postcondition P01
stateSchema SS2
output O1
outputSchema OS1
precondition PRE2
changeSchema CS3
postcondition P02
```

Figure 9.30: General proof skeleton for the semiformal specification.

9.4.4 Step 4

Isabelle Skeleton

Using the GPSa created in the previous step for the semiformal autopilot specification. The ZMathLang toolkit can then automatically generate an Isabelle skeleton

which translates the GPSa into Isabelle syntax. This is shown in figure ??.

```

theory gpsaln2
imports
Main

begin

record SS1 :: (*DECLARATIONS*)

locale ln2 =
fixes (*GLOBAL DECLARATIONS*)
begin

definition PRE1 :: "(*PRE1_TYPES*) => bool"
where
"PRE1 (*PRE1_VARIABLES*) == (*PRECONDITION*)"

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

definition P01 :: "(*P01_TYPES*) => bool"
where
"P01 (*P01_VARIABLES*) == (*POSTCONDITION*)"

definition O1 :: "(*O1_TYPES*) => bool"
where
"O1 (*O1_VARIABLES*) == (*EXPRESSION*)"

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

```

```

definition PRE2 :: "(*PRE2_TYPES*) => bool"
where
"PRE2 (*PRE2_VARIABLES*) == (*PRECONDITION*)"

definition CS3 :: "(*CS3_TYPES*) => bool"
where
"CS3 (*CS3_VARIABLES*) == True"

definition P02 :: "(*P02_TYPES*) => bool"
where
"P02 (*P02_VARIABLES*) == (*POSTCONDITION*)"

end

```

Figure 9.31: The Isabelle skeleton for the semiformal autopilot specification.

9.4.5 Step 5

Isabelle Skeleton filled in

The ZMathLang toolkit can now fill in the Isabelle skeleton using the ZCGa and ZDRA annotated document. The ZMathLang can only translate the parts of the specification which are written formally and have been annotated. Figure ?? shows the Isabelle skeleton filled in as much as automatically is possible. Since the specification isn't completely filled in, this is as far as we can get with the translation.

```

theory 1n2
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 PRE1 :: (*PRE1_TYPES*) => bool
where
"PRE1 (*PRE1_VARIABLES*) == (*PRECONDITION*)"

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

definition P01 :: (*P01_TYPES*) => bool
where
"P01 (*P01_VARIABLES*) == (*POSTCONDITION*)"

definition off_eng :: mode_status => bool
where
"off_eng mode == (mode = off ∨ mode = engaged)"

definition att_cwsDo :: 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

```

Figure 9.32: The Isabelle skeleton filled in for the semiformal autopilot specification.

9.5 Conclusion

In this chapter we have taken a two specifications and shown the entire path from the raw specification to it's translation in Isabelle. We have shown the L^AT_EX 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 generated 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 modulereg 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 10

Analysis

The concept for this thesis was to develop a new method to enable a stepwise approach to prove formal specifications instead of proving specifications in one step. The vending machine example has been fully proved in Proof Power Z (PPZed) and the birthday book example has been fully proved in Hol-Z (Hol-Z) in one step. There have not been any proofs for our examples already done in Isabelle (apart from the proofs I have already written myself which would be unfair to compare). We will now compare these examples to the proofs done in a stepwise method using ZMathLang.

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. Other variations are described in [?], [?] and [?].

Note: we use number of lines to compare the lengths of the proof and not to measure the efficiency or superiority over one proof over another.

To compare proofs we have looked at the amount of complexity in the specification. For example the vending machine example uses integers as ‘types’ in the specification. Where as other examples such as the birthdaybook declare its own ‘type’ within the specification and is used throughout. The complexity of proofs are discussed more in the next chapter. This chapter focusses on what ‘expertise’ is needed to create a proof for a specification using different theorem provers.

In figure ?? to prove the proof obligations (red parts) one must have some theorem prover expertise. However, helpful automated theorem proving tools which exist

such as ‘sledgehammer’ or ‘blast’. These tools are also available to a theorem proving expert. Therefore effort is the same. The difference is in translating the specification into Isabelle from L^AT_EX. One can translate the specification by hand-rewrite the entire specification again into Isabelle syntax or one can use ZMathLang for the translation with doing ZCGa and ZDRa checks along the way. This allows the user to ‘digest’ the specification and moreover check that the specification isabelle ZCGa and ZDRa correct before inputting into the theorem prover. This does not save time but allows checks to be done at an earlier phase instead of directly translating the entire specification into a theorem prover and finding grammatical and logical errors at the end. The user will then need to go over the entire specification and find where those errors have occurred.

10.1 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	L ^A T _E X partially automated into Isabelle	fl = 3 ep = 124 (63 automated)

Table 10.1: The vending machine proof using PPZed versus the ZMathLang proof.

Table ?? shows a comparison between the vending machine proof done in PPZed [?] and the vending machine proof done using the ZMathLang method (see appendix ??). 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 L^AT_EX 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 L ^A T _E X (including Z symbols)	some (optional)	yes
Knowledge of how to input specification into theorem prover	yes	no

Table 10.2: 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 ???. 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.

10.1.1 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.

10.1.2 Knowledge of theorem prover

In table ??, it states that a ‘little’ amount of theorem prover knowledge 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.

10.1.3 Knowledge of L^AT_EX

The translation path using the ZMathLang methods assumes the user already knows how to write a Z specification using L^AT_EX. The user then annotates these specifications using the annotations in the ZMathLang style package. The L^AT_EX 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 [?].

The schema boxes and symbols are written in PPZed’s own syntax. PPZed also has a user interface, (PPXPP), which 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 L^AT_EX 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 L^AT_EX file using

the PPZed extended character set. However to typeset the proof the instructions say that some familiarity with L^AT_EX is required.

10.1.4 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 L^AT_EX.

10.2 Birthday Book Example

The birthday book example (shown in appendix ??), was created by Spivey [?]. 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	L <small>A</small> T <small>E</small> X partially automated into Isabelle	fl = 8 ep = 120

Table 10.3: The birthday book proof using Hol-Z versus the ZMathLang proof.

Table ?? shows the comparison between the birthday book proof done using Hol-Z [?] and the birthday book proof done using the ZMathLang method (see appendix ??). 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 L^AT_EX 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 L ^A T _E X	yes	yes
Knowledge of how to input specification into theorem prover	some (sml into Hol-Z)	no

Table 10.4: Expertise needed for one step proof in PPZed and multi step proof using ZMathLang.

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

Since starting this thesis the Hol-Z project is no longer developed. Therefore the comparison done here is limited with the support available.

It is important to note that the time it takes translating a Z specification into a theorem prover with full proofs may or may not be longer using small steps in ZMathLang Vs using Hol-Z, as that will depend on the user and their expertise/ typing skills/ knowledge of Z etc. However what table ?? does show is that for both methods the user would need knowledge of Z and of L^AT_EX. For the actual translation of the specification alone (without proofs) the user would need none to very little knowledge of the theorem prover using ZMathLang but will need some knowledge of the theorem prover using Hol-Z. Using Hol-Z the user would also need to type up the specification again whereas in Z-MathLang the user would need to annotate the specification in L^AT_EX. The time it takes to do this again would depend on the user and if they are comfortable with the Hol-Z tool or L^AT_EX annotations. Therefore the ‘efficiency’ would be the same for both methods however using the

Hol-Z method the user would need additional Hol-Z and L^AT_EX expertise whereas using ZMathLang the user would need to have just L^AT_EX expertise on it's own. This is for the translation on it's own.

To complete the proofs for the specification the user would need both L^AT_EX and some theorem prover expertise. Therefore the 'efficiency' is about the same.

10.2.1 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 L^AT_EX for the system. However by using the ZMathLang method when annotating the Z specification in ZCGa, to 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.

10.2.2 Knowledge of theorem prover

Table ?? 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. 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 ??), 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.

The Hol-Z proof comes in 10+ files written in languages (hol-z, zeta, sml etc), the ZMathLang proof includes:

1. 1 ZCGa Labelled tex document + it's pdf output
2. 1 ZDRa Labelled tex document + it's pdf output

3. GPSa
4. proof skeleton in Isabelle
5. Filled in skeleton
6. Full proof in Isabelle

Both theorem provers offer a L^AT_EX written specification. The ZMathLang toolkit also produces graphs in order to help the user understand the layout and dependencies of the specification. Hol-Z does not offer such diagrams and therefore understanding the proof purely via code may be more difficult.

An e-mail from ‘Burkhart Wolff’ (who wrote the Hol-Z tutorial) explained the following about Hol-Z:

‘The port of the library is not particularly difficult though. What is hairy, is the front-end, and in particular the Zeta Parser. A project around 2012 trying to port it to a more recent Isabelle-version and a new Z parser/typechecker front-end was abandoned.’

10.2.3 Knowledge of L^AT_EX

In both methods the user will need to have the same amount of knowledge of L^AT_EX. This is because in both cases, the user will need to input their specification using L^AT_EX. 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, Z sections etc). In both cases the user will need to know how to import a package into L^AT_EX and then read the instructions in either case on the annotations which need to be used.

10.2.4 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 L^AT_EX, 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 to 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 L^AT_EX 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 theorem 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.

10.2.5 Comparison with similar tools for Z

Hol-Z [?] also analyses the Z notation, and 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 L^AT_EX 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. Hol-Z works differently to ZMathLang as it is a proof environment for Z specifications and will only analyse the parts of the document written in Z syntax, whereas ZMathLang will check any parts of the document which have been annotated by the user using the ZCGa and ZDRA annotations. Hol-Z uses Zeta [?] as a Z type checker.

Zeta [?] is an open environment for the development, analysis and animation of Z specifications. The system is aware of dependencies between the units and

attempt to exploit the units when they are changed. Zeta is different to the ZCGa type checker because the ZCGa will read the annotations written by the user which could include some informal text as well as Z syntax. The ZCGa did not aim to be a strong type checker like Zeta as the ZCGa intends to check the grammatical correctness of the specification at a high level, where the weak types can be checked when a specification is in the process of being developed. The Z specification would be strongly type checked once the user would have translated the specification into Isabelle (step 6 of the ZMathLang path.)

Fuzz [?] is Mike Spivey's type-checker for Z specifications. It includes style files for L^AT_EX and checks for the logical correctness and Z type correctness of Z specifications. It is a strong type checker for Z which takes Z specifications written in L^AT_EX as input. This is similar to the Zeta checker as it checks the specific "Z Types" and whether the correct symbols are used. This is different to the ZCGa type checker as the ZCGa checks for weak types and is aimed at early soft type checking (as full type checking is done once the specification is translated into Isabelle). The ZCGa . Therefore the grammatical correctness of partially formal specification can also be checked. The ZMathLang framework presented in this thesis uses the 'zed' L^AT_EX style package to typeset the Z specifications in the documents.

Proofpower [?] is a suite of tools supporting specification and proof in Higher Order Logic (HOL) and in the Z notation. Proofpower contains 6 packages:

1. **PPDev** - The ProofPower developer kit, mainly comprising SLRP, a parser generator for Standard ML.
2. **PPTex** - The ProofPower interface to TeX and L^AT_EX.
3. **PPXpp** - The X Windows/Motif front-end for ProofPower.
4. **PPHol** - The HOL specification and proof development system.
5. **PPZed** - The Z specification and proof development system.
6. **PPDaz** - The Compliance Tool for specifying and verifying Ada programs.

Proofpower Z is a theorem proving environment specific for Z notations. PPZed has its own user interface to input specifications. Users can input specifications using Proofpower syntax or an extended character set in which the user can click on the characters required. The type checking and proofs are all done within the Proofpower interface.

Proofpower syntax is more similar to a normal drawing of a schema than the way we have translated schemas to Isabelle using ZMathLang. This is because Isabelle is a generic theorem prover for all of mathematics whereas Proofpower is made specific to Z specifications. The aim of ZMathLang was to incorporate all specifications and not just ones written in Z. Again proofpower has incorporated type checking like Zeta and Fuzz which is also done in Isabelle using the ZMathLang method. However as mentioned before the ZCGa is a weak type checker which checks grammatical categories at a higher level than a usual type checker.

10.3 Conclusion

This section compares two 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). This chapter also compared other similar tools which have been implemented to analyse Z specification. We have identified that although these tools are similar- none are identical to the ZMathLang toolkit. 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 obligations of the specification are a whole research area on their own and beyond the scope of this thesis but touched upon in chapter ???. The ZMathLang aspects and it's tools can be used as helpful means to the user digest the specification and allow them to understand the syntax of their specifications.

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

Chapter 11

Evaluation and Discussion

In this chapter we go through a few case studies and discuss the difference between the specification translations if any. Table ?? 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 tool-kit. In this chapter we take one example from each class (Steamboiler from appendix ??, ModuleReg from chapter ?? and AutoPilot from appendix ??) 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 11.1: A table showing the specifications we have translated into Isabelle using ZMathLang

We have categorised 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. These categories are important as the examples using sets and terms are more complex than the specifications only using terms. This is because there is an extra ZCGa type to manage. The specification which are categorised as "incomplete" can be using terms and sets however they are not full specification so have not been fully translated into Isabelle and/or have proofs to check the stateInvariants. 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 it was difficult to find full examples of functional specification and therefore we couldn't add it to the list of examples.

11.1 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 L^AT_EX specification file, without any annotation. Then we discuss the complexity of the ZCGa annotated specifications and ZDRa annotated specifications and how this affects the translation into Isabelle.

11.1.1 Raw Latex Count

Table ?? shows 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 L^AT_EX the raw specification has.

Specification	Environment				Lines of L ^A T _E X
	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 11.2: How many zed, schema and axdef environments and lines of L^AT_EX code makes up each specification

We list information about how many different environments and lines of L^AT_EX make up each specification in table ???. 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}... \end{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 L^AT_EX there is then the total amount of environments increase. However, there are three exceptions to this trend. The ‘*BirthdayBook*’ specification, ‘*ClubState2*’ specification and ‘*Vending 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 ??.

However, there are also many lines which can not be simplified in the specification for example if a specification uses a certain Z-type i.e. ‘COLOUR’ then ‘COLOUR’ must be declared at the beginning of the specification: [COLOUR] in the specification this line can not be removed. Therefore we use the number of lines plus the number of environment to discuss the complexity.

The number of environments is also used to discuss complexity as we could have simple program which only has one function (such as a program which outputs a number). This type of specification would only have 1 output schema and therefore be considered as quite simple. Other programs/specifications can have many different functions and therefore many different environments. Hence, when we discuss complexity in this chapter the more the environments a specification has the more complex it is.

11.1.2 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 11.3: 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 ???. 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 ‘*SteamBoiler*’ 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 introduced 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 than `declarations` and `expressions` combined as `schemaText` contains all `declarations`, `expressions` and `Schem-aNames` 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}}\\
\res \set{\{\term{lec}\}}})} \leq \term{\set{maxPlaces}^{\set{maxPlaces}}}}
```

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 ??.

The SteamBoiler specification has by far the most ZCGa weak types in total at 1204. Therefore in terms of complexity it would take the longest to annotate in ZCGa. In terms of doing a ZCGa weak type check it would also take the longest as the ZCGa would have to parse through all 1204 weak types and check they are correct. There are only 4 ‘sets’ in the SteamBoiler specification but in our case this doesn’t make a difference because the ZCGa weak typing rules are similar for set and term. Therefore if the next complex specification (ProjectAlloc) still wouldn’t class as being more complex than the Steamboiler specification. GenDB can also be argued as being complex as it contains the highest amount of weak types **and** containing a definition. However as the specification only contains 1 definition out of a total of 378 weak types, we can not dispute that GenDB has a strong case for being the most complex. This conclusion is drawn just by analysing the amount of ZCGa weak types. We will look at ZDRa in the next section.

11.1.3 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 ?? and give details of the amount of relations in each specification in table ??.

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 11.4: How many of each ZDRa instances exists in each specification.

Table ?? shows our example specifications in left column and the following ZDRa instances:

A	axiom	TS	totaliseSchema
SS	statesSchema	PRE	precondition
IS	initialSchema	PO	postcondition
CS	changeSchema	O	output
OS	outputSchema	SI	stateInvariants

From table ?? we can see that all specifications have either 1 or 2 statesSchema'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. Invariants can be relied upon to be true throughout the duration of a computer program. Therefore to prove the program is correct it is important to check that the stateInvariants still hold true throughout every transformation on the current state of the program.

Transformations on the state and represented by changeSchemas in Z. Therefore we check the stateInvariants still hold true after each changeSchema in ZMathLang.

Some specifications in our examples do not have any stateInvariants such as our autopilot specification. In this case we can not prove that our stateInvariants are consistent throughout the specification. However in this was we can prove that all preconditions must have a corresponding postcondition or output. Therefore we can say:

Lemma 11.1.1. *precondition* \longrightarrow *postcondition* \vee *output*

In [?], Ed Currie states that predicates which state what must be true about the ‘before’ state of the system and the inputs, in order for the operation to take place. These are known as *preconditions*.

If an operation can be either a *postcondition* or an *output*. Then we can say lemma ?? holds. For our sample of specifications we can see evidence of this in table ???. There are more combined postconditions and outputs then there are preconditions. 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. The only way we could have more preconditions then combined outputs and postconditions would be in an incomplete or semi-formal specification.

Specification	ZDRa Relations				
	initialOf	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 11.5: How many of each ZDRa relations exists in each specification.

We can cross reference the table showing the amount of instances (table ??) with the table showing the relations (table ??). For example, the relation *initialOf* can only occur if the specification has an *initialSchema*. Not all specifications have an *initialSchema* and therefore do not have an *initialOf* relation.

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 the third largest specifications from our examples (in terms of lines of LATEX) in our sample set however it has quite a small amount of relations.

11.2 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.

We have chosen these particular case studies in order to analyse the differences in complexity between those specifications using just terms and those specifications using both terms and sets.

11.2.1 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. The steamboiler specification is the largest from our examples (we use Beckert extended version from Steam Boiler Control) It is made up of 507 lines of LATEX code, 10 zed environments, 34 schemas and 3 axiom definitions. When annotating with ZCGa there were 297 schematext, 26

declarations, 282 expressions, 595 terms and 4 sets. When annotated with ZDRA there were 6 axioms, 2 stateSchema's, 2 initialSchema's, 21 changeSchema's, 6 outputSchema's, 6 totaliseSchema's, 21 preconditions, 23 postconditions, 12 outputs and 1 set of stateInvariants.

11.2.1.1 Natural Lanaguge Specification of the Steamboiler

The steam boiler itself is a water level and steam quantity measuring device, with four pumps and four pump controllers. There is a valve for emptying the boiler.

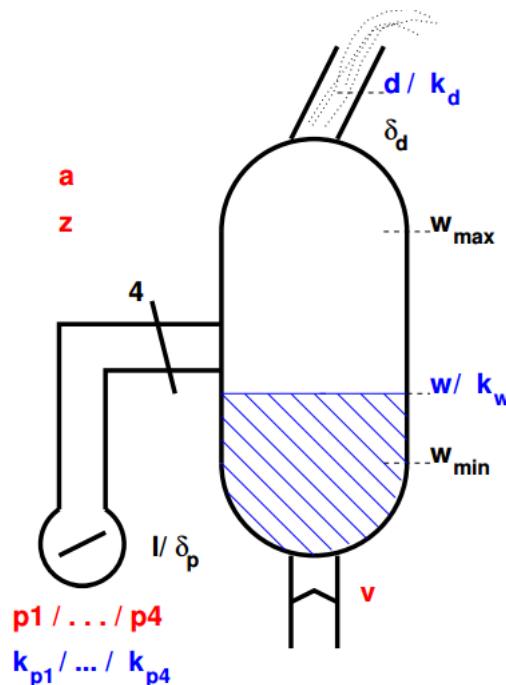


Figure 11.1: A diagram showing a theoretical Steamboiler.

An example of how the steamboiler could look is shown in figure ???. The variables of the steamboiler are shown in table ??.

variables	description
w_{min}	minimal water level
w_{max}	maximal water level
l	water amount per pump
d_{max}	maximal quantity of steam exiting the boiler
δ_p	error in the value of the pumps
δ_d	error in steam
w	water level
d	amount of steam exiting the boiler
$k_{p,i}$	pump i works/broken
k_w	water level measuring device works/broken
k_d	steam amount measuring device works/broken
p_i	pump i on/off
v	valve open/closed
a	boiler on/off
z	state init/norm/broken/stop

Table 11.6: The variables of the steamboiler and their descriptions.

The full formal specification for the steamboiler is 10 pages long which can be found in [?]. Therefore we have given small examples taken from the full specification.

11.2.1.2 ZMathLang steps for the steamboiler case study.

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

\begin{document}

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

\begin{zed}
OnOff ::= on | off
\end{zed}

\begin{zed}
OpenClosed ::= open | closed
\end{zed}

Physical Constants

\begin{axdef}
w_{min}: \nat \\
w_{max}: \nat \\
w_{opt}: \nat \\
l: \nat \\
d_{max}: \nat \\
\delta_p: \nat \\
\delta_d: \nat \\
w_{min} < w_{max}
\end{axdef}

Measured values

\begin{schema}{Input}
w?: \nat \\
\end{schema}

\begin{zed}
Pumps
p_1, p_2, p_3, p_4 : OnOff
\end{zed}

\begin{zed}
SteamBoiler0
Pumps
v : OpenClosed
a : OnOff
z : State
\end{zed}

Control values

\begin{zed}
PumpsOff
Pumps'
p'_1 = off \wedge p'_2 = off \wedge p'_3 = off \wedge p'_4 = off
\end{zed}

\begin{zed}
PumpsOn
Pumps'
p'_1 = on \wedge p'_2 = on \wedge p'_3 = on \wedge p'_4 = on
\end{zed}

Auxiliary Schemata

```

Figure 11.2: The formal specification L^AT_EX code for the steamboiler system.

We show the L^AT_EX code for part of the raw steamboiler specification in figure ?? and it's pdflatex counterpart in figure ??.

We then annotate the specification using ZCGa and ZDRA labels.

Figure 11.3: The formal specification for the steamboiler system.

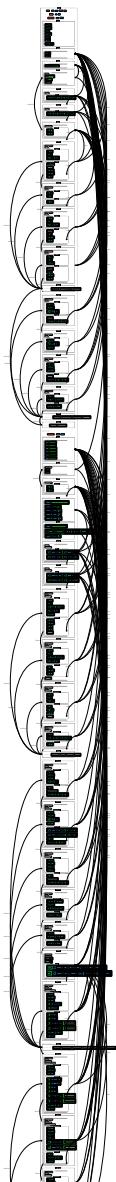


Figure 11.4: An example of the original steamboiler specification annotated in ZCGa and ZDRA.

Messages
Spec Grammatically Correct
Messages
Warning! Specification not correctly totalised
Specification is Rhetorically Correct

Figure 11.5: The result when checking the steamboiler specification with the ZCGa and ZDRA checkers.

Since we only have a warning and no errors when checking the steamboiler specification we can now generate a goto graph and dependency graphs for it.

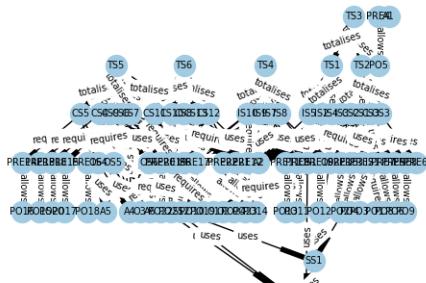


Figure 11.6: The dependency graph produced for the steamboiler specification.

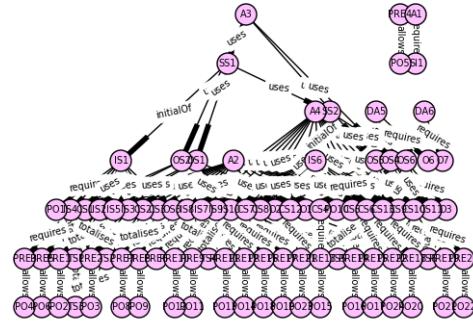


Figure 11.7: The goto graph produced for the steamboiler specification.

The dependency and goto graphs are shown in figures ?? and ?? respectively. Since there are a lot of ZDRA instances and therefore a lot of nodes, both the dependency graph and goto graph are cluttered. One way to solve this is to use zooming facilities for graphs. However, as the program at the moment outputs the graphs as ‘.png’ files, the program will need to be modified in a way so that the graphs are produced in a different file format for graph exploration. We will discuss this as a limitation in the next section.

From the goto graph the ZMathLang tool kit automatically generates a general proof skeleton, which uses the order from the goto graph to order the instances in how they should appear in any theorem prover. Part of the skeleton for the steamboiler specification is shown in figure ??.

```

axiom A1
stateInvariants SI1
axiom A2
axiom A3
stateSchema SS1
initialSchema IS1
postcondition P01
changeSchema CS7
precondition PRE8
postcondition P09
changeSchema CS2
precondition PRE2

```

Figure 11.8: Gpsa for the steamboiler specification.

We can now translate the Gpsa into Isabelle syntax using the ZMathLang tool-

kit.

```

theory steamboilerSkelton
imports
Main

begin
(*DATATYPES*)

record SS1 =
(*DECLARATIONS*)

locale ln2 =
fixes (*GLOBAL DECLARATIONS*)
assumes SI1
begin

definition IS1 :: 
"(*IS1_TYPES*) => bool"
where
"IS1 (*IS1_VARIABLES*) == (P01)"

definition CS7 :: 
"(*CS7_TYPES*) => bool"
where

```

```

definition TS3 :: 
"(*TS3_TYPES*) => bool"
where
"TS3 (*TS3_VARIABLES*) == (*TS3_EXPRESSION*)"

end

record SS2 = SS1 +

```

```

definition IS2 :: 
"(*IS2_TYPES*) => bool"
where
"IS2 (*IS2_VARIABLES*) == (P010)"

definition OS5 :: 
"(*OS5_TYPES*) => bool"
where
"OS5 (*OS5_VARIABLES*) == (04)"

definition OS4 :: 
"(*OS4_TYPES*) => bool"
where
"OS4 (*OS4_VARIABLES*) == (03)"

```

Figure 11.9: Part of the isabelle skeleton for the steamboiler specification.

Part of the isabelle skeleton for the steamboiler specification is shown in figure ???. Since the steamboiler example has 2 stateSchema's the ZMathLang tool-set creates 2 isabelle records in the theory file. The top left image shows the beginning part of the isabelle skeleton, where the first stateSchema (or record) sets the state of the theory. Midway down the theory file the first record ends and a new one is added with the line record SS2 = SS1 +. Towards the end the isabelle skeleton there are lemma's to check the consistency for all state changing schema's (CS) in the format described in chapter ?? section ???. Using the ZCGa annotated specification and the steamboiler isabelle skeleton, the ZMathLang tool support can now fill in the isabelle skeleton the declarations, expressions, schemaNames etc.

```

theory steamboilerProof
imports
Main

begin
datatype State = init | norm | broken0 | stop
datatype OnOff = on | off
datatype OpenClosed = open0 | closed
datatype WorksBroken = works | broken

record SteamBoiler0 =
PSWITCH :: "State"
W_MAX :: "nat"
D_MAX :: "nat"
PAMOUNT :: "State"
W_MIN :: "nat"
A :: "OnOff"
DELTA_D :: "nat"
DELTA_P :: "nat"
L :: "nat"
V :: "OpenClosed"
Z :: "State"
W :: "nat"

lemma (in thesteamboiler) SNormalStop0_L1:
"(∃ steamboiler0 :: SteamBoiler0.
 ∃ a' :: OnOff.
 ∃ steamboiler0' :: SteamBoiler0.
 ∃ w_max' :: nat.
 ∃ w_min' :: nat.
 ∃ z' :: State .
 ∃ v' :: OpenClosed.
 (z = norm)
 ∧ (w < w_min ∨
 w > w_max)
 ∧ (a' = off ∧
 z' = stop)
 → (w_min < w_max
 ∧ (w_min' < w_max')))"
sorry

lemma (in thesteamboiler) SInitStop0_L2:
"(∃ steamboiler0 :: SteamBoiler0.
 ∃ a' :: OnOff.
 ∃ steamboiler0' :: SteamBoiler0.
 ∃ w_max' :: nat.
 ∃ w_min' :: nat.

```

Figure 11.10: Part of the filled in isabelle skeleton for the steamboiler specification.

In figure ?? we show 3 parts of the filled in isabelle skeleton (halfbaked proof).

The first part shows the beginning of the halfbaked proof which initiates the beginning of the proof. Since *SS1* in this case was the root of the tree in the goto graph it sets **SteamBoiler0** as the first record. Half way through the theory file we see another record, **SteamBoiler1** which was *SS2* in ZDRA. This is shown in the bottom picture in figure ???. *SS2* introduced 2 new state variables, *S* and *delta* which are added to the new record. Towards the end of the halfbaked proof, ZMathLang has filled in the lemma's to prove which are sanity checks for the specification. The sanity checks for the steamboiler specification check that all 21 of the steamboiler ‘*changeSchema’s*’ do not conflict with the stateInvariants. So after every change in state $w_{min} < w_{max}$. It fills in the lemma’s with the correct syntax so that the user only needs to delete the word ‘*sorry*’ prove the properties in order to get a proof of their specification.

```

lemma (in thesteamboiler) SNormalStop0_L1:
"(∃ steamboiler0 :: SteamBoiler0.
 ∃ a' :: OnOff.
 ∃ steamboiler0' :: SteamBoiler0.
 ∃ w_max' :: nat.
 ∃ w_min' :: nat.
 ∃ z' :: State .
 ∃ v' :: OpenClosed.
 (z = norm)
 ∧ (w < w_min ∨
 w > w_max)
 ∧ (a' = off ∧
 z' = stop)
 → (w_min < w_max
 ∧ (w_min' < w_max')))"
by (smt State.distinct(9))

lemma (in thesteamboiler) SInitStop0_L2:
"(∃ steamboiler0 :: SteamBoiler0.
 ∃ a' :: OnOff.
 ∃ steamboiler0' :: SteamBoiler0.
 ∃ w_max' :: nat.
 ∃ w_min' :: nat.

```

Figure 11.11: Manually proven lemma for the steamboiler specification.

Using the lemma's which have been generated in figure ?? we have proved all of these lemmas for the steamboiler specification, part of which is shown in figure ???. By doing so, we have now proven that non of the state changing schemas conflict with the state invariants of the specification. To do this we have manually deleted the ‘*sorry*’ command, used the Isar tool ‘*sledgehammer*’ which has indicated that to prove this particular lemma (shown in figure ???) it can be proven by `smt State.distinct(9)`. Therefore it is true that the ‘`SNormalStop0`’ schema does not conflict with the state Invariants. We did this step manually for all remaining lemmas, the full proof of the steamboiler specification can be found in [?].

In [?] a link between safety requirements, event-B models and corresponding fragments of a safety case is created. Laibinis et al present a method and tool support linking formal modelling in Event-B and safety case construction. Whereas ZMathLang translates formal modelling into the Isabelle theorem prover which can then be used as evidence to create a safety case for high integrity systems. Laibinis paper employs refinement to create proofs (point 5 from figure ??) whereas ZMathLang employs sanity and consistency checks (points 1 and 2 from figure ??). Refinement proofs requires refinement specifications to be present in order to develop the proofs.

The aim of ZMathLang was to be able to create proofs for specifications which are also semi-formal and therefore creates proofs sanity and consistency instead.

In [?] SteamBoiler example, Laibinis et al uses an abstract specification as well as 4 other refinement specifications. ZMathLang can translate an abstract specification and creates proof obligations without a refinement specification. Safety requirements are added and translated into event-B. An example of such a safety requirement is ‘*during system operation the water level shall not exceed the predefined safety boundaries*’. In ZMathLang additional safety requirements to prove can also be added and written in Isabelle.

One advantage of using [?] to prove the SteamBoiler specification is that the SteamBoiler has been written in Event-B which is supported by the Rodin platform to prove refinements and that this technique has been used by industry. However a disadvantage of using this technique is that in order to create proofs one must have refinement specifications as well as an abstract specification. The specification must also be a completed finished product. An advantage of the ZMathLang toolkit is that one can do proofs on a single abstract specification, and the proofs can also be done on specification which aren’t complete and ones which are semi-formalised. Another advantage of using ZMathLang is that the steamboiler can undergo other checks (ZGa, ZDRa) just using the abstract specification. So the specification doesn’t need refinement specifications. The diagrams and results produced (dependency graphs, GoTo graphs etc) can also be presented as evidence within the safety case. However, to get this evidence the user will need to label the specification which could be argued as a little tedious. A disadvantage of using ZMathLang to prove the Steamboiler specification is to generate a solid safety case one needs to add safety related requirements in Isabelle.

11.2.2 Case Study 2: A specification using both terms and sets.

This case study based is on the *ModuleReg* specification which uses both terms and sets. The specification has been translated into Isabelle using the ZMathLang

framework. The entire ZMathLang works for the ModuleReg example is shown in chapter ??.

The *ModuleReg* specification is our smallest example with 43 lines of L^AT_EX code, 1 *zed* environment and 3 *schema*'s. There are 20 labels of *schemaText*, 6 *declarations*, 18 *expressions*, 13 *terms*, and 31 *sets*. Since there are *stateInvariants* for the *modulereg* specification, ZMathLang was able to generate lemma's to prove for the 2 *changeSchemas*. There is also 1 *stateSchema*, 2 *preconditions* and 2 *postconditions*. There are 3 *requires* relations, 2 *allows* and 2 *uses*.

Since the *modulereg* specification is quite small but did have *stateInvariants* which ZMathLang could prove are satisfied throughout the specification, we decided it would be a could example to show the full workings of. This is shown in chapter ??.

In conclusion there is not much difference in the difficulty of translation between those specifications using just terms and those specifications using both terms and sets.

11.2.3 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 [?] 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 "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.

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 11.12: An example of the original Autopilot specification.

11.2.3.1 ZMathLang steps for the autopilot case study.

We give the informal specification in figure ?? and one which we are beginning to formalised in figure ???. We have highlighted in red the parts which we have formalised in figure ???. 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 ?? and ?? respectively. Once annotated we can check the annotated document for ZCGa and ZDRa errors.

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 | press_cas_eng | press_alt_eng |
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 | engaged

<i>off_eng</i>
<i>mode : mode_status</i>
<i>mode = off ∨ mode = engaged</i>

AutoPilot

Figure 11.13: An example of the Autopilot specification partially formalised.

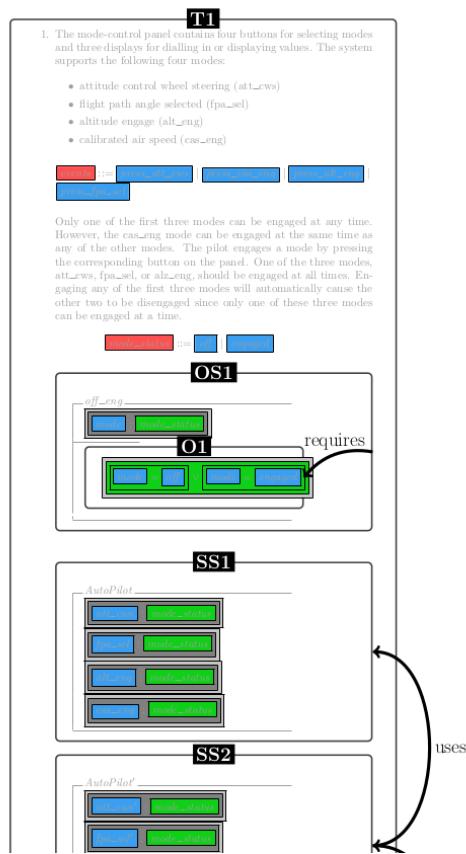


Figure 11.14: An example of the original Autopilot specification annotated in ZCGa and ZDRa.

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 ?? and ??). When checking with ZDRa we have a warning message telling the user that the specification is not correctly totalised. We remind the reader that a specification is ‘*not correctly totalised*’ when every precondition has a corresponding postcondition or output (section ??). This does not matter for now in our semi-formal specification as we can still carry on with the translation.

When checking the specification for ZDRa, ZMathLang has also produced a dependency graph and goto graphs (shown in figures ?? and ??):

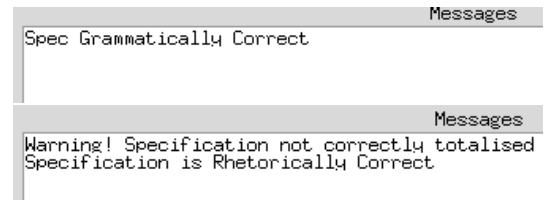


Figure 11.15: The result when checking the autopilot specification with the ZCGa and ZDRa checkers.

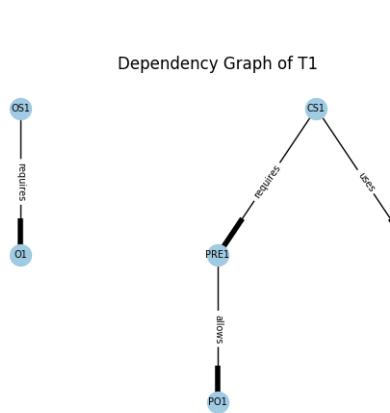


Figure 11.16: The dependency graph produced for the autopilot specification.

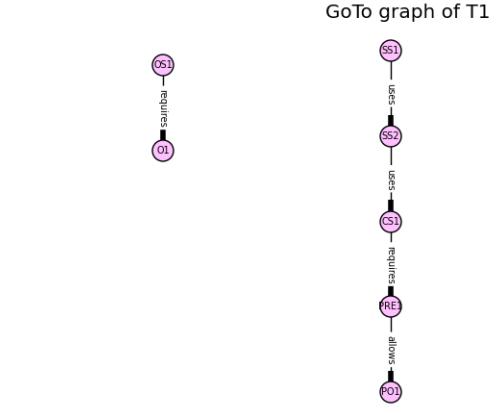


Figure 11.17: The goto graph produced for the autopilot specification.

With the dependency graph (figure ??) 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 separate as they do not have any relations which any parts of the main tree, the only dependency they have is on each other where *O1* requires *OS1*.

We can say that the dependency graph *describes* the relation between instances and the goto graph (figure ??) *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 ??). 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 separate. 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 would 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*. Therefore *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 P01
```

Figure 11.18: 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 ???. 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 consistency across the specification and thus need state invariants to be present.

```

theory gpsaln2
imports
Main

begin
(*DATATYPES*)

record SS1 =
(*DECLARATIONS*)

locale 1n2 =
fixes (*GLOBAL DECLARATIONS*)
begin

definition OS1 :: 
"(*OS1_TYPES*) => bool"
where
"OS1 (*OS1_VARIABLES*) == (0"

definition CS1 :: 
"(*CS1_TYPES*) => bool"
where
"CS1 (*CS1_VARIABLES*) == 
(PRE1)
^ (P01)"

end
end

```

Figure 11.19: The Isabelle skeleton produced for the autopilot specification.

```

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 = 

definition att_cwsDo :: 
"mode_status ⇒ mode_status ⇒ mode_st
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' = engaged) "

```

Figure 11.20: The autopilot specification in Isabelle syntax.

ZMathLang can automatically translate the Gpsa into Isabelle syntax (figure ??), this is now an Isabelle skeleton. The Isabelle skeleton has not yet taken the ZCGa information as one can get to this step with just the ZDRA annotated document. Once the Isabelle is filled in (figure ??) we have the annotated specification in Isabelle form. This can now give the user an idea of how to input their specification into Isabelle syntax, without them having prior knowledge of Isabelle. It is important to note that this is as far as the ZMathLang translation goes. Since there are no state Invariants with this case study no lemma's to check for consistency have been generated. The user can add the state Invariants in their raw L^AT_EX specification, or fully formalise their specification. Another way to fully prove their specification is to add other properties to the Isabelle document.

11.3 Analysing examples

In this section we analyse the examples we have successfully translated into Isabelle and proved the sanity of the specification.

We remind the reader of figure ?? in chapter ???. ZMathLang is able assist the user with the translation of specification up to the point where sanity properties are produced but not proven. In our largest case study (section ??) the user did not have to look through all the state changing schema's and write the sanity checks for all of them. The properties were already generated for each changeSchema however, the user did have to go through each property and prove it. In total, there were 21 changeSchema's and 1 set of stateInvariants, therefore there was 21 properties which the user had to prove manually.

11.3.1 ModuleReg

The modulereg is one of our smallest examples, however with 1 set of stateinvariants and 2 changeSchemas, ZMathLang automatically produces 2 lemma's to check the sanity of the specification. An example of one of these lemmas is shown in figure ???. This is one of the lemma's automatically generated and thus we have the '*sorry*' command at the end to show that it needs manual input from the user to complete the proof.

```

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 ∈ students)
     $\wedge$  (m ∈ degModules)
     $\wedge$  ((p, m) ∉ taking)
     $\wedge$  (taking' = taking ∪ {(p, m)})
     $\wedge$  (students' = students)
     $\wedge$  (degModules' = degModules))
     $\wedge$  (Domain taking ⊆ students)
     $\wedge$  (Range taking ⊆ degModules)
     $\wedge$  (Domain taking' ⊆ students')
     $\wedge$  (Range taking' ⊆ degModules'))"
  sorry"
```

Figure 11.21: An example of one of the lemma's to check for consistency in the modulereg specification.

To prove this lemma we remove the ‘*sorry*’ command or put our cursor at the end of the lemma ready to input our methods to start the proof. In this case ‘*Auto sledgehammer*’ again found a proof using the ‘cvc4’ SMT solver (shown in figure ??). With this lemma we are aiming to prove the sanity of the specification where the changeSchema `RegForModule` does not conflict with the stateInvariants either before or after the state has been changed.

```

proof (prove): depth 0
goal (1 subgoal):
  1.  $\exists$ degModules students taking p degModules' students' taking' m.
     (p ∈ students  $\wedge$ 
      m ∈ degModules  $\wedge$ 
      (p, m) ∉ taking  $\wedge$  taking' = taking ∪ {(p, m)}  $\wedge$  students' = students  $\wedge$  degModules' = degModules)  $\wedge$ 
      Domain taking ⊆ students  $\wedge$ 
      Range taking ⊆ degModules  $\wedge$  Domain taking' ⊆ students'  $\wedge$  Range taking' ⊆ degModules'
Auto Sledgehammer ("cvc4") found a proof: by (smt Domain_empty Domain_insert Range_intro Range_empty Range_
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').
```

Figure 11.22: Output shown when proving the lemma ‘`RegForModule`’ shown in figure ?? .

By clicking on the auto solving method shown in figure ?? we can now complete the proof for the `RegForModule_L1` lemma. This is shown

```
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')
```

Figure 11.23: The ‘RegForModule’ lemma proved using Auto sledgehammer methods.

The second lemma in the moduleReg specification we managed to prove using ‘blast’ thus having a complete proof for the complexity of the modulereg specification.

We can see that the complexity of the proof used for the RegForModule_L1 lemma in the modulereg specification is larger than the complexity of the proof for the SNormalStop0_L1 in the steamboiler specification because it uses more tactics. There are other ways to measure complexity of a proof (e.g. type of tactics used, time taken to prove etc). However for the purpose of this thesis we say the more tactics we need to use for a proof the more complex it is. Although we used ‘Auto sledgehammer’ to assist proving the lemma’s there are 16 methods used in proving the RegForModule_L1 lemma (Domain_empty, Range_empty etc.) compared with 1 method used in proving the SNormalStop0_L1 lemma (State.distinct(9)). Again we can say that there might of been an alternate way to prove these particular lemma’s however we have chosen to prove them in this way to show variation. Since there are more state changing schema’s in the steamboiler specification there are also more lemma’s to prove with the steamboiler then there is in the modulereg

specification to obtain a fully proven specification which checks the complexity of the system.

11.3.2 Vending Machine

The vending machine example has 3 state changing schemas (shown in table ??) however since it does not have any labeled stateInvariants, ZMathLang can not automatically produce any properties to prove the consistency of the specification. If we refer back to figure ?? in chapter ?? it shows that the ZMathLang tool-kit goes slightly past the point of '*specification in isabelle with no proof*' however the automation of ZMathLang can only go past that point **if** there are changeschema's and stateInvariants labelled. Otherwise the ZMathLang tool-kit can only translate the specification into isabelle syntax with no lemma's or properties to prove. Thus it is up to the user to carry on manually inputting their properties to obtain a fully proven specification.

11.3.3 Other examples

Each specification has a different amount of lemma's which the user needs to prove depending on how many 'stateInvariants' and 'changeSchema's' there are. If the specification does not have any stateInvariants then ZMathLang will not produce any lemmas to prove for the sanity of the specification.

Specification	Amount of generated lemmas	Total amount of tactics
Steamboiler	20	41
ProjectAlloc	5	18
VideoShop	3	3
TelephoneDirectory	4	8
ClubState	4	4
ZCGa	6	6
GenDB	4	13
Timetable	4	4
BirthdayBook	1	1
AutoPilot	0	0
ClubState2	3	4
Vending Machine	0	0
ModuleReg	2	18
Total	56	120

Table 11.7: A table to show the amount of automatically generated lemmas and total amount of tactics used for each specification.

We show the amount of lemmas and total amount of tactics needed to prove the sanity of each specification in table ???. Some lemma's need only one tactic to be proved whilst other need a few. There are various different ways to prove these lemmas and it all comes down to the personal preference of the user. We have mainly used Isabellas '*sledgehammer*' tool to assist us with our proving. We see that the AutoPilot specification has 0 generated lemmas and therefore 0 tactics to prove them. This is because the AutoPilot specification is only semi-formalised and there will need to be *preconditions*, *changeSchemas* and *stateInvariants* to exist for the lemmas to be automatically generated. Once the user adds these ZDRA categories to the Autopilot specification in a formal manner (using Z) then the lemmas can be generated.

Proof complexity is a whole different research area that studies the concept of

complexity from the point of view of logic [?]. It is connected with computational complexity, but the goals are different. There are many ways to measure ‘*complexity*’ i.e. the size of the proofs, how strong theory is needed to prove the theorem etc. Pudlak [?] describes more on the complexity of proofs. In order to discuss the complexity of proofs in this thesis we will look at the size of the proof for each lemma.

11.3.3.1 ProjectAlloc

The projectAlloc specification [?] has 5 changeSchema’s and 1 set of stateInvariants therefore it has 5 lemmas which ZMathLang has automatically generated to check for the consistency. To prove these lemmas we have in total used 18 tactics.

The lemmas to check that the changeSchema’s did not conflict with the stateInvariants with were as follows:

```
\<exists> variables and types.

Preconditions

\<and>

PostConditions

\<longrightarrow>

(((dom studInterests) \<inter> (dom lecInterests) = {}))

\<and> (dom allocation \<subseteqq> dom studInterests)

\<and> (ran allocation \<subseteqq> dom lecInterests)

\<and> (dom maxPlaces = dom lecInterests)

\<and> (\<forall> lec \<in> dom maxPlaces.

(card ({l. the (allocation l) = lec})) \<leq> the (maxPlaces lec))

\<and> ((dom studInterests') \<inter> (dom lecInterests') = {})

\<and> (dom allocation' \<subseteqq> dom studInterests')

\<and> (ran allocation' \<subseteqq> dom lecInterests')

\<and> (dom maxPlaces' = dom lecInterests')

\<and> (\<forall> lec \<in> dom maxPlaces').

(card ({l. the (allocation' l) = lec})) \<leq> the (maxPlaces' lec)))"
```

Our most complex lemma to prove is the `AddLecturer_L5` lemma which we used 9 tactics:

```
(metis, full_types, dom_empty, dom_empty, dom_empty,
dom_eq_singleton_conv, dom_restrict, inf.idem and insert_not_empty).
```

The most simple lemma to prove was `AddStudent_L3` and `DeAllocate_L2` where we just used 1 tactic (`fastforce` and `auto` respectively).

11.3.3.2 VideoShop

The videoShop specification [?] has 3 changeSchemas and 1 set of stateInvariants, therefore ZMathLang has generated 3 proof obligations to check that none of the changeSchemas conflict with the stateInvariants and stateInvariants prime.

The structure of lemma's for the videoshop specification are as follows:

```
\<exists> variables and types.

Preconditions

\<and>

PostConditions

\<longrightarrow>

(Domain rented \<subseteqq> members)

\<and> (Range rented \<subseteqq> dom stockLevel)

\<and> (\<forall> t \<in> Range rented.

    card ({p. (p, t) \<in> rented}) < (the (stockLevel t)))

\<and> (Domain rented' \<subseteqq> members')

\<and> (Range rented' \<subseteqq> dom stockLevel')

\<and> (\<forall> t \<in> Range rented'.

    card ({p. (p, t) \<in> rented'}) < (the (stockLevel' t))))"
```

We are able to prove all 3 lemma's by the tactic `blast`.

11.3.3.3 TelephoneDirectory

The telephone directory ?? has 1 set of stateInvariants and 4 changeSchema's. Therefore we have 4 lemma's which ZMathLang generated and which we need to

prove. The structure for the lemmas to check the consistency of the telephone directory specification are as follows:

```
\<exists> variables and types.

Preconditions

\<and>

PostConditions

\<longrightarrow>

((dom phoneNumbers = Domain persons)

\<and> (dom phoneNumbers' = Domain persons'))"
```

These sanity check make sure that when updated the telephone directory that the people listed in the domain of phoneNumbers is equal to the list in the domain of persons.

To prove the first 2 lemmas `AddPerson_L1` and `RemoveNumber_L2` we only needed to use a single tactic (`auto` and `fastforce`) respectively. However the last 2 lemmas (`RemovePerson_L3` and `RemoveNumber_L4`) required 3 tactics each. For example to prove the `RemovePerson_L3` lemma we needed to use `smt`, `Diff_insert_absorb` and `mk_disjoint_insert`. In total we used 8 tactics to prove all 4 properties.

11.3.3.4 ClubState

The clubstate specification [?] has 4 changing schemas and thus 4 properties to check the state invariants are not conflicted when the state has been changed. The stateinvariants for the clubstate specification is are as follows:

```
\<exists> variables and types.

Preconditions

\<and>

PostConditions

\<longrightarrow>

(hall \<subseteqq> badminton)

\<and> (card hall \<leq> maxPlayers)
```

```
\<and> (hall' \<subseteqq> badminton')
\<and> (card hall' \<leq> maxPlayers))"
```

Here we wish that all the people in the hall must be members of badminton and the number of players can not exceed the maximum amount both before the change and after the change in state. The first 2 lemma's in our clubstate specification (`LeaveHall_L1` and `AddMember_L2`) were proven by `auto` and the last two properties (`EnterHall_L3`: and `RemoveMember_L4`) were proven by `blast`.

11.3.3.5 ZCGa

In the specification [?] representing the ZCGa we have 6 properties to prove. The syntax for the properties which ZMathLang has generated are:

```
\<exists> variables and types.

Preconditions

\<and>

PostConditions

\<longrightarrow>

(TermDeclaration \<subseteqq> declarations)

\<and> (SetDeclaration \<subseteqq> declarations)

\<and> (dvars \<subset> sets \<union> terms)

\<and> (sets \<inter> terms = {})

\<and> (TermDeclaration' \<subseteqq> declarations')

\<and> (SetDeclaration' \<subseteqq> declarations')

\<and> (dvars' \<subset> sets' \<union> terms')

\<and> (sets' \<inter> terms' = {}))"
```

Since we have 6 changing schemas we need to check that none of the schemas conflict with the stateInvariants before and after the state has been changed. In this proof, we have proven 2 lemmas

(`CorrectExpression_L1` and `CorrectConstantTerm_L5`) by `smt` and the remaining 4 using `blast`.

11.3.3.6 GenDB

GenDB [?] has 4 consistency checks we must prove against the stateInvariants. The syntax for the stateInvariants are as follows:

```
\<exists> variables and types.

Preconditions

\<and>

PostConditions

\<longrightarrow>

(Domain parent \<union> Range parent \<subseteqq> dom sex
\<and> (\<forall>p :: PERSON. (p, p) \<notin> parent^*)
\<and> (\<forall>p :: PERSON. \<forall>q :: PERSON.
\<forall>r :: PERSON. ({(p,q),(p,r)} \<subseteqq> parent)
\<and> q \<noteq> r \<longrightarrow>
the (sex q) \<noteq> the (sex r))

\<and> (Domain parent' \<union> Range parent' \<subseteqq> dom sex'
\<and> (\<forall>p :: PERSON. (p, p) \<notin> parent'^*)
\<and> (\<forall>p :: PERSON. \<forall>p :: PERSON.
\<forall>r :: PERSON. ({(p,q),(p,r)} \<subseteqq> parent')
\<and> q \<noteq> r \<longrightarrow>
the (sex' q) \<noteq> the (sex' r))"
```

Out of the 4 lemmas we proved, 2 lemmas can be proven by the tactic `blast`. The other 2 lemmas we proved by using 5 other tactics. For example, we proved the `AddPerson_L3` property using the tactics `metis`, `mono_tags`, `lifting`, `empty_iff` and `rtrancl.rtrancl_refl`. This made sure that when a person has been added to the genDB that the preconditions and postconditions satisfied the stateInvariants.

11.3.3.7 Timetable

The timetable specification [?] has 4 schemas in which the original state has changed and 1 set of stateInvariants that must be obeyed throughout the specification. The stateInvariants for the timetable specification are as follows.

```
\<exists> variables and types.

Preconditions

\<and>

PostConditions

\<longrightarrow>

(\<forall> s \<in> dom studentTT. \<forall> m \<in> dom moduleTT.

(the (studentTT s) \<inter> the (moduleTT m) \<noteq> empty)

\<longrightarrow>

(dom (the (studentTT s))) moduleTT m = (dom (the (studentTT s)))))

\<and> (\<forall> s \<in> dom studentTT'.

\<forall> m \<in> dom moduleTT'.

(the (studentTT' s) \<inter> the (moduleTT' m) \<noteq> empty)

\<longrightarrow>

(dom (the (studentTT' s))) moduleTT' m = (dom (the (studentTT' s))))"
```

There are 4 properties to prove in the timetable specification. We have been able to prove 3 properties (`RegForModule_L1`, `AddStudent_L2` and `ScheduleModule_L4`) using `smt` and 1 property `DescheduleModule_L3` using `blast`.

11.3.3.8 BirthdayBook

Since we used the simplest form of the birthdaybook ?? specification (there are many different forms) we only had 1 lemma to prove. The `AddBirthday_L1` property was proven by `auto` however we could have also solved it using `smt`.

```
\<exists> variables and types.

Preconditions

\<and>

PostConditions

\<longrightarrow>

(known = dom birthday)

\<and> (known' = dom birthday'))"
```

11.3.3.9 ClubState2

The clubstate2 specification [?] is an extension of the original clubstate specification. Since the specification stemmed from the original clubstate we have to make sure that the changeState schemas do not conflict with the stateInvariants from the first stateSchema as well as the stateInvariants in the ClubState2 state schema. We have the following stateInvariants from the ClubState2 specification:

```
\<exists> variables and types.
```

Preconditions

```
\<and>
```

PostConditions

```
\<longrightarrow>
```

```
((hall \<subseteqq> badminton)
\<and> (card hall \<leq> maxPlayers)
\<and> [onCourt, (Range waiting)] partition [hall]
\<and> (hall' \<subseteqq> badminton')
\<and> (card hall' \<leq> maxPlayers))
\<and> [onCourt', (Range waiting')] partition [hall']"
```

The stateInvariants taken from the ClubState2 state schema is

`[onCourt, (Range waiting)] partition [hall]` however the ClubState2 state schema uses the ClubState schema. Therefore the original stateInvariants

`((hall \<subseteqq> badminton) and (card hall \<leq> maxPlayers))` must also be checked.

The 3 lemmas which needed to be proved use a total of 4 tactics. The first and last lemmas are proven by `auto` where as the second lemma (`LeaveHall_L2`) is proven using 2 tactics (`smt` and `empty_iff`).

11.3.3.10 ModuleReg

The ModuleReg specification has 2 changeSchemas and 1 set of stateInvariants. therefore ZMathLang has produced 2 lemmas which we need to prove.

```
\<exists> variables and types.

Preconditions

\<and>

PostConditions

\<longrightarrow>

(Domain taking \<subseteqq> students)

\<and> (Range taking \<subseteqq> degModules)

\<and> (Domain taking' \<subseteqq> students')

\<and> (Range taking' \<subseteqq> degModules'))"
```

The AddStudent_L2 property we managed to prove using a single tactic (**blast**) however to prove the first lemma (RegForModule_L1) we used 17 tactics, these were:

```
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'
```

11.3.4 Summary

In conclusion we can see our examples how each of the lemma's are proven in order to check the sanity of the specification. The amount of tactics used in the specification does not depend on the amount of lemma's generated. The complexity of the lemma can be measured in a variety of different ways [?] and therefore some specification will require more or less strategies to prove (or more complex strategies). For example the Module Reg specification has 2 lemmas but requires 18 tactics to be proved. Whereas the Timetable specification has double the amount of lemma's (4) and only requires 4 tactics to prove the specification. Therefore the complexity of the specification can not be measured in terms of "number of lemmas". Finding proofs for these lemma's has to be done manually (or using the automated help already in Isabelle). Some specification will require more theorem prover expertise (moduleReg) in order to complete the final step to have a fully proven specification. Whereas others (Timetable) will require less theorem prover expertise to prove the automatically

generated lemmas. Therefore the amount of effort for the last step of ZMathLang is difficult to measure as it depends on the specification to be proven.

11.4 Reflection and Discussion

To check the consistency (section ??) of all our specification we had to prove 56 lemmas with a total of 120 tactics. In this section we discuss how far the ZMathLang tool-kit can take us, how difficult it is for the user to get the full proof and what other proof is left.

11.4.1 How far can ZMathLang tool-kit take us and what is left for the user to do manually.

Looking back at our examples described in the previous section. The more complex the language of the specification (whether it uses terms and set or just terms), the more complex the proofs and syntax of the proofs are. The vending machine has no stateinvariants therefore ZMathLang does not generate any lemmas to be proven. This does not mean that stateInvariants are good or bad, it means that the ZMathLang toolkit will be unable to generate any lemma to check for the consistency of the specification without them. The steamboiler specification, which is our longest example, uses only terms but has 2 stateSchemas and therefore 2 records. We were able to prove all the lemmas using 2 different automatic isabelle tools (blast and sledgehammer).

We remind the reader of figure ?? in chapter ??, `sledgehammer` and `blast` are the two tools which have the most automated proving power. With the manual proof to complete the theorem the user should have some basic knowledge of how to prove lemmas in Isabelle, at least have some basic knowledge of the Isabelle tools available to assist them in proving the lemmas. If the software/system designer is not an Isabelle expert and has got up to step 5 in the ZMathLang steps then they would have the three following options:

1. Learn no Isabelle at all and pass on the proofs to be completed by an Isabelle

expert.

2. Learn Isabelle in depth until they gain enough knowledge to prove their specification and add more proofs if needed.
3. Learn the basic automated proving tools (highlighted in figure ??) to prove all (if not then some) of the lemmas, then hand the rest to an Isabelle expert to finish off.

In the first case we will need 2 people to complete the proof. Since the system designer has already translated the specification into Isabelle syntax automatically so a lot of the work has been done. The theory file has been set up and properties have been written, the expert will just need to prove them. Therefore a big chunk of work has been done. With this approach the majority of the work goes to the system designer as they do not need to learn anything beyond their existing knowledge. However, there needs to be an Isabelle expert ready to complete the proofs and depending on the syntax of the specification this may be difficult to prove. There also might be other properties which the stakeholders of the project may wish to be proven. Again, to add these extra properties, a little bit more Isabelle knowledge is needed.

In the second scenario we will need only 1 person to complete the proof (the system designer). Using the ZMathLang tool-kit will assist the designer translating the specification into Isabelle syntax, however the user will then need to learn more Isabelle to finish off the proof for their specification. Since the ZMathLang has already produced an Isabelle file the user can then learn the Isabelle syntax by example as well as reading through the various Isabelle documentations online. In this scenario there is only one person doing the entire proof, there is less cost in finding another Isabelle expert and paying them to complete the proof (depending on the specification). The system engineer has to spend more time on learning Isabelle syntax and proving in Isabelle which may become long and tedious. The stakeholders may also want other properties proven about the specification which again is down to the system designer to learn.

The third case is a compromise between the first two scenarios, it will require the system designer and on some occasions and an Isabelle expert. The system engineer translates the specification into Isabelle using the ZMathLang tool-kit to assist them. They then learn the very basic automated proving tools, to assist them in proving the lemmas to check for consistency. By doing this the system engineer may be able to prove the entire specification for consistency using the automated tool. However some lemmas if they are more complex may require more Isabelle skills than just the automated Isabelle tools to prove. Therefore the system engineer will be required to prove as much as they can and if there are some parts which can not be proved with automation, they can pass this along to an Isabelle expert. The Isabelle expert will then have less work to prove the remainder of the proof. If the stakeholders require more properties proved then the Isabelle expert can do this as well.

In the third case an Isabelle expert may not be required and proving the specification *could* be done by one person (depending on the specification) if the stakeholders only want to check the system for consistency and the syntax of the specification is relatively easy where automated tools can be used to prove the specification. However, depending on the complexity of the specification, an Isabelle expert may be needed sometimes if the system designer can not prove the entire specification with just the automated tools and/or if the stakeholders require a more rigorous proof.

Automation on both sides are constantly being redeveloped and revisited. One can automate a specification into Isabelle covering formal methods and perhaps one day, informal specification could be translated. On the other end, there are many SMT solvers being developed to cover more and more existing lemmas and theorems. However, due to the variations in systems, syntax and properties the user may wish to prove, it is difficult to cover everything. The more complex the system, the less automation available to assist in the users proofs. For a simple system, there exists automation on both sides to cover the proofs. However, once the user wishes to build a more complex system, the user will then need to learn more about their chosen theorem in order to check for complete correctness.

11.4.2 Assumptions and limitations of the ZMathLang tool-kit

The recommendations made in this thesis are based on several assumptions and limitations as to what may be done to translate a Z specification into Isabelle using our ZMathLang tool-kit.

The assumptions are as follows:

- Firstly, we like to point out that the ZMathLang can only translate specifications with syntax used in our examples. Since the language of mathematical notation is vast we can not incorporate the entire syntax of mathematics in the ZMathLang within the scope of this thesis. More explanation is given on this topic in the next chapter.
- The ZMathLang tool-kit uses L^AT_EX and is implemented in python2.7. It uses the following python packages:
 - **re:** To read the users annotations from ZCGa and ZDRa.
 - **networkx:** To implemented the instances and relationships into a graph form.
 - **matplotlib:** To produce the goto and dependency graphs.
 - **Tkinter:** To run the user interface.

We assume that a user wanting to use the ZMathLang tool-kit has installed the correct packages.

- One of the assumptions the tool-kit makes is that if the specification holds 1 state schema (and thus 1 record) then all the lemma's produced are under that record. Thus, if the specification holds 2 or more records then the lemmas produced will need to define which record they belong to. For example in the SteamBoiler specification we manually input '(in thesteamboiler)' after the word '**lemma**'. This does not have to be done to specification which only hold one state schema.

- Another assumption the ZMathLang tool-kit makes is that the document it is checking contains a single specification. Although the ZCGa can check multiple specification in a single document, the rest of the tools assume that it is checking 1 system. Thus, when producing the dependency and goto graphs and the specification isn't linked, it will assume that the instances are under one specification. Also when the ZDRa translates the specification into Isabelle it will do so in a single theorem document. More discussion on this is in the 'Future work' section in the next chapter.
- We also assume that the user actually wishes to check the specification for consistency. As Stepney discussed in [?] there are many properties to prove in a system. It is down to the stakeholders and clients of the project. For example a high integrity system may have some universal guidelines or codes of conduct it must obey. Proving properties to check for consistency is highly valuable, however other properties such as if preconditions hold, or relationship between two specification may also be wanted by the project stakeholders. These other properties may be added at the end of stage 6 and if the consistency checks are not needed, they can be deleted from the Isabelle file manually.

We like to highlight the limitations of the ZMathLang tool-kit. They are as follows:

- One of the limitations of the ZMathLang tool-kit is in the way the ZDRa checks if the specification is correctly totalised. Although an incorrectly totalised specification can still be translated into Isabelle a warning message comes up for incorrectly totalised specifications. If the users totalises preconditions e.g `\totalises{TS#}{PRE#}` and all preconditions have been totalised then no warning appears. However, if the user totalises schemas which hold preconditions within them e.g. `\totalises{TS#}{CS#}` then the ZDRa checker doesn't pick up on it and would say the specification is incorrectly totalised.
- Another limitation is in the way the dependency and goto graphs are presented. If the specification is relatively small, with not a lot of instances and

connections then the graphs are presented quite clearly. On the other hand, when a specification is more complex with many instances the graph presents the nodes bundled together. Visually this becomes difficult to see the relationship between the nodes, it would be better if the nodes were represented with larger spaces between them.

11.5 Expert Reviews

We gave experts from various fields and industries an overview of the MathLang framework and asked for their opinion on it. The following questions were asked:

1. Would you use ZMathLang toolkit for any of your work?
2. Do you see the ZMathLang toolkit being useful somewhere in your work? If yes where and what steps in particular?
3. Is breaking up the translation path easier/harder/same with ZMathLang toolkit and why?
4. Any other comments.

The responses we got from the various industry expert were the following:

The first two steps of ZMathLang would be useful in ordering the requirements in a logical order, this would greatly help the customer. The specifications we write are for systems which have a quick turnaround therefore we don't usually do proofs on our specifications as it is too expensive both in manpower and costs so the last steps would be redundant.

Bid Manager @ QinetiQ

As a Bid Solution Architect working in the Aerospace and Defence domain handling a multitude of requirements can be a highly complex, especially when developing solutions and concepts that comprise of systems of systems. Steps 0 to 3 of ZMathLang would facilitate the ordering of specific requirements quickly and effectively. Furthermore, the dependency graph is extremely useful for tracking changes of a solution, whereby it would enable System Engineers to easily identify the significance of a proposed change and the knock-on implications, thus allowing for more expedient and accurate cost estimating. If ZMathLang can be deployed easily with short user implementation time, steps 0 to 3 would be beneficial throughout complex programmes.

Bid Solution Architect @ Thales

The ZMathLang Toolkit appears to be an useful program. However at the moment all controller specifications are written by engineers and then checked and approved by two other senior/principal engineers. There is no room for language and logical errors. I see the subjected toolkit useful when autonomous vehicles would be introduced on our roads. New, more complicated programs and controller specifications would be expected to be developed to assure an appropriate connection is kept between vehicles approaching to the junctions/pedestrian crossings and controllers. The ZMathLang could then be used to assist users to analyse and highlight any logical and language errors, inconsistencies and mistakes especially if the new controller specifications/other programs would require more than one person working on it especially from different teams.

Traffic Signal Engineer @ AECOM

Aviation engineering encompasses a large range of disciplines where the ZMathLang toolkit would provide assistance to work carried out by both myself and colleagues at conceptual and detailed stages of design. A few examples of where the toolkit could be used are as follows; Conceptual design of airfields entails a series of well organised requirements from clients with processes enabling designers to make logical and objective decisions. Steps 0-2 of the toolkit would be particularly useful within this work. However design does, to a certain degree, require human interference in the process. The ZMathLang toolkit could provide useful automation to the surface access issues within an airfield masterplan. This would include the automation of passenger buses from airport terminal to aircraft stands. All steps within the toolkit would provide a complete analysis and successful implementation. Whilst as an aviation engineer working on airfield design specifically, I believe the toolkit would have further benefits to the wider aviation industry. Further automation in the industry is sought as technology is continuously improving. I would consider the toolkit could play an important part in the improvement of baggage handling systems, discussions around the automation of air traffic control services, and an airline's approach to their operations with regards to aircraft movements and the associated efficiencies.

Aviation Engineer @ AECOM

11.6 Conclusion

The ZMathLang tool-kit has been tried and tested on various examples of Z specifications. We have translated specifications with syntax using set and terms and specifications using terms only. It seems that the grammatical types do not affect the ease of translation, but the amount of instances and the notations of mathematics do. The Steamboiler specification was our most complex example and we had to manually input extra parts to the lemmas in order to prove them. The amount of changeSchema's and stateInvariants affect how many properties the tool-kit will produce. The complexity of the mathematical notation will determine how difficult the properties are to prove, whether they can be proven with the assistance of

automatic tools or whether they need more Isabelle expertise to prove.

We have outlined some of the assumptions and limitation of the ZMathLang tool-kit and we discuss the achievements of the system and future work in the next chapter.

Chapter 12

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 section ?? we give add concluding thoughts to this thesis.

12.1 Achievements of this thesis

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

1. Staged an approach to translating semi-formal and formal specifications into Isabelle with automatic assistance.
2. Built a collection of tools to enable a step by step approach for Z specifications.
3. Formalised and proved properties on a number of examples of Z specifications.
4. Demonstrated and evaluated the performance of the tool set on a convincing set of examples.

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

The first accomplishment of this thesis is also the general aim of this thesis. The step by step method is outlined in chapter ??, which outlines how a user can get from a Z specification to a full proof in Isabelle. An example of this on a single specification is given in chapter ???. Each of these steps are described 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 ??). A L^AT_EX ZMathLang package has also been created in order to display ZMathLang annotations when a L^AT_EX specification is compiled. This L^AT_EX package is also what is used by the ZCGa and ZDRa python program in order to parse through the specification. The ZMathLang L^AT_EX package is shown in appendix ??.

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 [?].

12.1.2 Built a collection of tools to enable a step by step approach for Z specifications.

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

One limitation of the ZCGa checker is that the user needs to annotate their specifications by hand using the L^AT_EX package. This may sometimes be a repetitive and boring task and improvement to this limitation is described in section ???. 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 ???.

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

Like the ZCGa, the ZDRa annotations for the specification has to be done by the user. A 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 be added to the current user interface described in chapter ???. A second limitation of the current ZDRa is that users can chunk any part of specification (formal or informal) the translation from the ZDRa annotated text can only be done from Z into Isabelle. It may be useful to translate from any formal specification into Isabelle (or any other theorem prover). More information on this extension is described in section ??.

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 are in total 5 items automatically produced 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 ??) show how the instances are related to each other, the Gpsa (chapter ??) show in which logical order the instances should be in order to be translated into a theorem prover with added instance to act as proof obligations. The Isabelle skeleton (chapter ??) uses the ZDRa instance names and creates a skeleton in Isabelle syntax. The halfbaked proof (chapter ??) is produced by using the Isabelle skeleton and the ZCGa annotated document. The filled in Isabelle skeleton is therefore the original specification translated in Isabelle syntax along with added proof obligations.

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

The proof obligations created in the Gpsa are properties which check the consistency 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 ??.

12.1.3 Formalised and proved properties on a number of examples of Z specifications.

The third contribution to this thesis is shown in chapter ??, chapter ??, appendix ??, ??, ?? and in [?] (as all the examples are quite long and including them will exceed the thesis limit). The ModuleReg example has been discussed from a raw specification to a fully proven specification with it's automatically generated lemmas. The lemma's have then been proven in the theorem prover Isabelle using various tactics. All our specification examples have been fully proven for consistency checks in Isabelle except for Vending Machine and Autopilot as they didn't have any State Invariants and therefore no lemma's where automatically generated for them.

One limitation of this contribution is that the lemma's we have proven only check the sanity and consistency of the specifications (see figure ?? for definitions) and there are many more checks one can do on a specification such as properties across specifications and emergent properties in one specification. One popular way to prove a specification is to create proofs between a specification and a refinement specification. However a refinement specification would be difficult to automate on it's own. One way of extended the ZMathLang toolkit would be to create lemma's

between a specification and it's refinement.

12.1.4 Demonstrated and evaluated the performance of the tool set on a convincing set of examples.

The final contribution to this thesis is we have analysed and evaluated the performance of the ZMathLang toolset. This contribution is discussed in chapters ?? and ???. We have also discussed three case studies in chapter ?? of how the toolkit works. The analysis discusses the difference between fully proven specifications using other methods Vs specifications proven using ZMathLang. The evaluation discusses the complexity of the specification and the advantages of using the ZMathLang toolkit as well as some of it's limitations. The complexity of the specifications as well as the complexity of the proofs are evaluated when using the ZMathLang toolkit.

A limitation of this contribution is that the evaluation only looked at the specifications and their proofs. Other ways to evaluate the specification could include the time it took to create the proofs and compare them to the time it took to complete the proof using another method. However this would be quite difficult as we would need to experiment using the same specification to do a fair comparison. In some examples we have used, they have not been proved using another method. The examples which have been proven using a different method (such as BirthdayBook in ProofPower-Z) are difficult to get figures for (such as how long the proof would take) as the ProofPower-Z community is no longer supported. Further work using the ZMathLang toolkit could include conducting experiments with theorem prover experts and participants not proficient with theorem proving and get their feedback on which method they prefer to prove specifications.

12.2 ZMathLang Current and Future Developments

12.2.1 Other Current Developments

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 targeted 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.

12.2.1.1 Other Developments

An extension to ZMathLang has started being developed by Fellar [?], [?] 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 [?], [?]. 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.

12.2.2 Future Developments

The future developments of ZMathLang have been discussed occasionally between students and supervisors during meetings. This section puts together and sum-

marises these ideas and presents them to the reader in order to provide a general idea of future developments.

12.2.2.1 Automisation of the annotation

At present, the user needs to annotate their formal specification by hand using L^AT_EX commands before being check by the various correctness checkers. This sometimes can be a time-consuming task especially if the user isn't familiar to L^AT_EX 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 L^AT_EX) 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 increase the workload for the user especially on very large specifications.

12.2.2.2 Extension to more complex proof obligations

The proof obligations described in this thesis are properties to check the consistency 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.

12.2.2.3 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 preferred by certain users and ultimately 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 translate 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 specifications 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.

12.2.2.4 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 ultimately translated fully into a theorem prover.

12.2.2.5 More than one system specification in one document

Occasionally, some systems are made up of lots of smaller subsystems. Or one may want to design a system specification which are unrelated to each other 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 terms¹ 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 separate 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 can add the a function in the ZDRa program to reset all instances and relationships if it sees a new specification or ‘\theory’.

12.3 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.

¹By correct terms we mean terms which are labelled with ZCGa annotations such as \term and are weakly correctly typed.

Chapter 13

Specifications in ZMathLang

The specifications translated using ZMathLang written here are the ones which are referred to in this thesis. The full range of examples of specifications can be found in [?].

13.1 Vending Machine

13.1.1 Raw Latex

```
\documentclass{article}
\usepackage{zed}
\begin{document}
\begin{zed}
price:\textsf{nat}
\end{zed}

\begin{schema}{VMSTATE}
stock, takings: \textsf{nat}
\end{schema}

\begin{schema}{VM\_operation}
\Delta VMSTATE \\
cash\_tendered?, cash\_refunded!: \textsf{nat} \\
bars\_delivered! : \textsf{nat}
\end{schema}

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

\begin{schema}{insufficient\_cash}
cash\_tendered? : \textsf{nat}
\where
cash\_tendered? < price
\end{schema}

\begin{schema}{some\_stock}
stock: \textsf{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}
```

13.1.2 Raw Latex output

VMSTATE _____
 $stock, takings : \mathbb{N}$

VM_operation _____
 $\Delta VMSTATE$
 $cash_tendered?, cash_refunded! : \mathbb{N}$
 $bars_delivered! : \mathbb{N}$

exact_cash _____
 $cash_tendered? : \mathbb{N}$
 $cash_tendered? = price$

insufficient_cash _____
 $cash_tendered? : \mathbb{N}$
 $cash_tendered? < price$

some_stock _____
 $stock : \mathbb{N}$
 $stock > 0$

VM_sale _____
VM_operation _____
 $stock' = stock - 1$
 $bars_delivered! = 1$
 $cash_refunded! = cash_tendered? - price$
 $takings' = takings + price$

VM_nosale _____
VM_operation _____
 $stock' = stock$
 $bars_delivered! = 0$
 $cash_refunded! = cash_tendered?$
 $takings' = takings$

$$VM1 \hat{=} exact_cash \wedge some_stock \wedge VM_sale$$

$$VM2 \hat{=} insufficient_cash \wedge VM_nosale$$

$$VM3 \hat{=} VM1 \vee VM2$$

13.1.3 ZCGa Annotated Latex Code

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

\begin{zed}
\text{\declaration{\term{price}: \expression{\nat}}}

\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}} \\
\text{\declaration{\term{bars\_delivered!}: \expression{\nat}}} \\
\end{schema}

\begin{schema}{exact\_cash}
\declaration{\term{cash\_tendered?}: \expression{\nat}} \\
\text{where} \\
\text{\expression{\term{cash\_tendered?}} = \term{price}} \\
\end{schema}

\begin{schema}{insufficient\_cash}
\declaration{\term{cash\_tendered?}: \expression{\nat}} \\
\text{where} \\
\text{\expression{\term{cash\_tendered?}} < \term{price}} \\
\end{schema}

\begin{schema}{some\_stock}
\declaration{\term{stock}: \expression{\nat}} \\
\text{where} \\
\text{\expression{\term{stock} > \term{0}}} \\
\end{schema}

\begin{schema}{VM\_sale}
\text{VM\_operation} \\
\text{where} \\
\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}

\begin{schema}{VM\_nosale}
\text{VM\_operation} \\
\text{where} \\
\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}

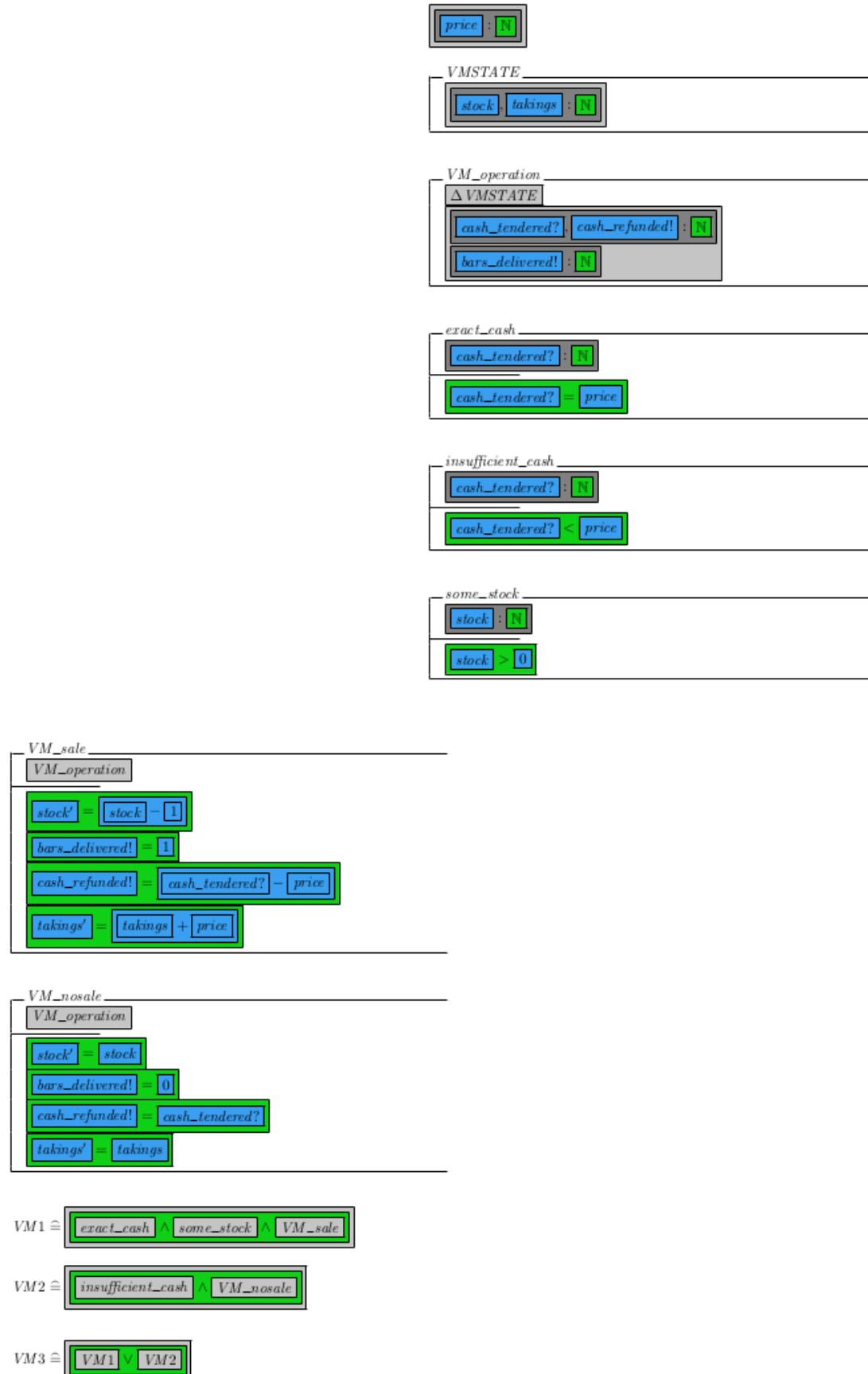
\begin{zed}
VM1 \def \text{\expression{\text{exact\_cash}}} \land \\
\text{\text{some\_stock}} \land \text{\text{VM\_sale}} \\
\end{zed}

\begin{zed}
VM2 \def \text{\expression{\text{insufficient\_cash}}} \land \\
\text{\text{VM\_nosale}} \\
\end{zed}

\begin{zed}
VM3 \def \text{\expression{\text{VM1} \lor \text{VM2}}} \\
\end{zed}

\end{document}
```

13.1.4 ZCGa output



13.1.5 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{P01}{stock' = stock - 1 \\
bars\_delivered! = 1 \\
cash\_refunded! = cash\_tendered? - price \\
takings' = takings + price}
\end{schema} }

\uses{CS1}{CS0}
\requires{CS1}{P01}

\draschema{CS2}{

\begin{schema}{VM\_nosale}
VM\_operation
\where
\draline{P02}{stock' = stock \\
bars\_delivered! = 0 \\
cash\_refunded! = cash\_tendered? \\
takings' = takings}
\end{schema} }

\uses{CS2}{CS0}
\requires{CS2}{P02}

\draschema{TS1}{

\begin{zed}
VM1 \def{ exact\_cash } \land some\_stock \land VM\_sale
\end{zed} }

\uses{TS1}{PRE1}
\uses{TS1}{PRE3}
\uses{TS1}{CS1}

\draschema{TS2}{

\begin{zed}
VM2 \def{ insufficient\_cash } \land VM\_nosale
\end{zed} }

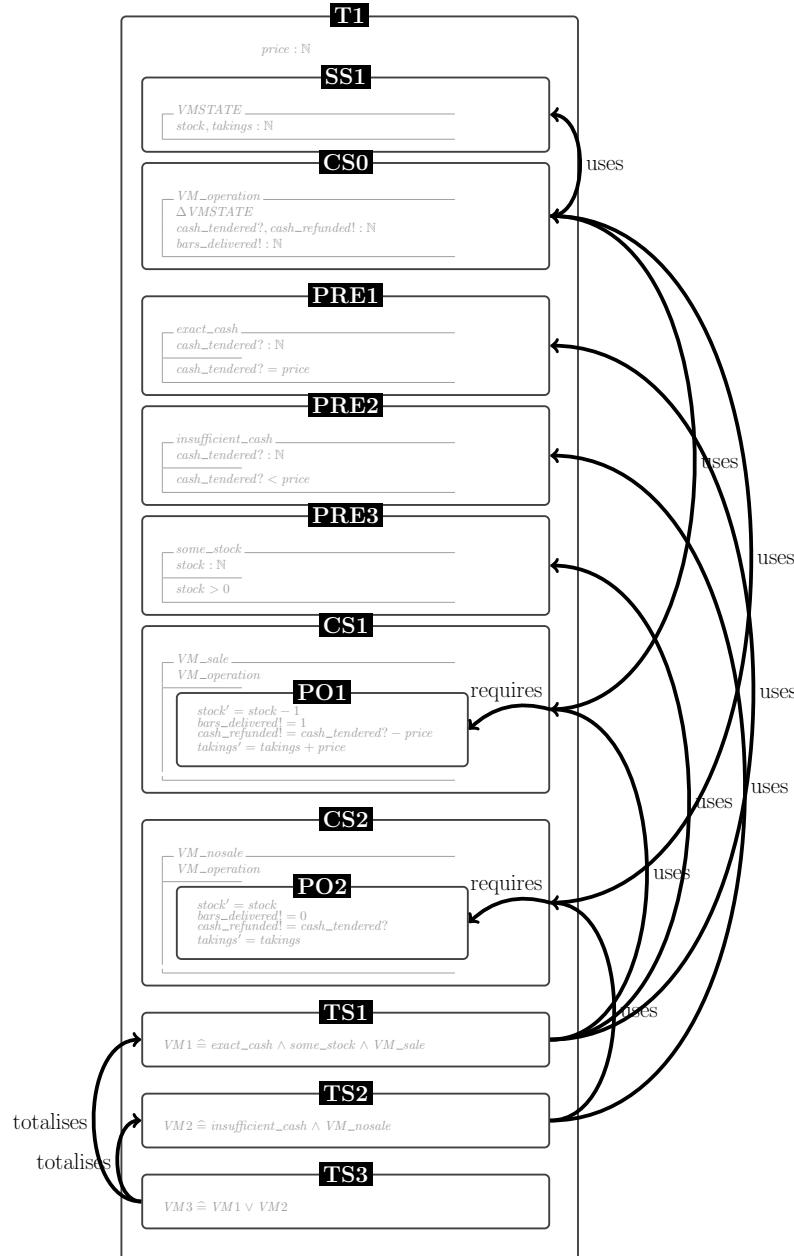
\uses{TS2}{PRE2}
\uses{TS2}{CS2}

```

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

\totalises{TS3}{TS1}
\totalises{TS3}{TS2}
}
\end{document}
```

13.1.6 ZDRa Output



13.1.7 ZCGa and ZDRa Annotated Latex Code

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

\drattheory{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{P01}{%
\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}{P01}

\draschema{CS2}{

\begin{schema}{VM\_nosale}
\text{VM\_operation}
\where
\draline{P02}{\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}{P02}

\draschema{TS1}{

\begin{zed}
VM1 \def \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 \def \text{\expression{\text{insufficient}\_cash}
\land \text{VM\_nosale}}}
\end{zed}

\uses{TS2}{PRE2}
\uses{TS2}{CS2}

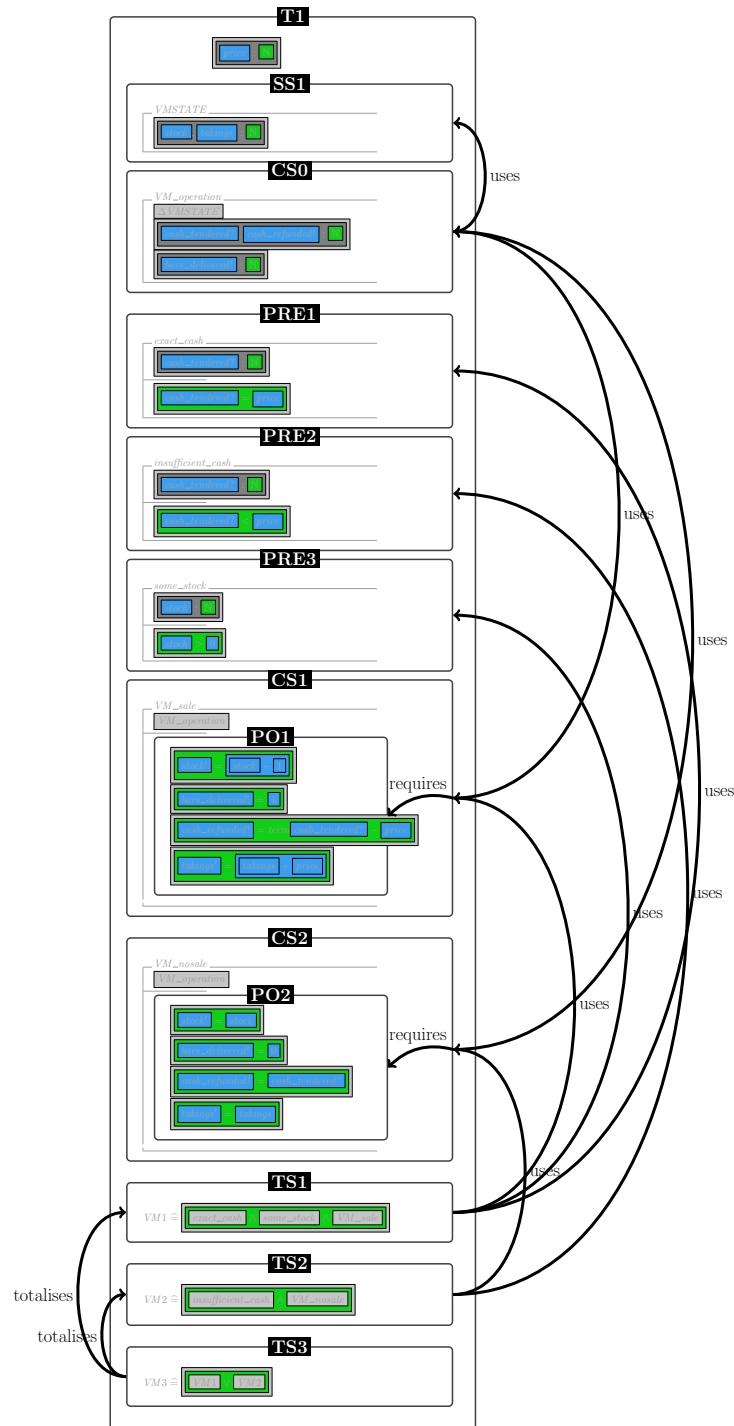
\draschema{TS3}{

\begin{zed}
VM3 \def \text{\expression{\text{VM1} \lor \text{VM2}}}
\end{zed}

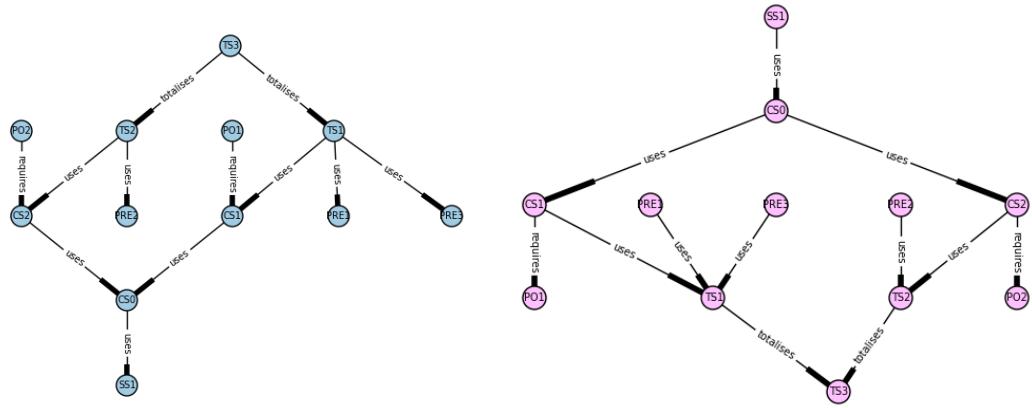
\totalises{TS3}{TS1}
\totalises{TS3}{TS2}
}
\end{document}

```

13.1.8 ZCGa and ZDRA Output



13.1.9 Dependency and Goto Graphs



13.1.10 General Proof Skeleton

```

stateSchema SS1
precondition PRE1
precondition PRE2
changeSchema CS0
precondition PRE3
changeSchema CS2
postcondition P02
totaliseSchema TS2
changeSchema CS1
postcondition P01
totaliseSchema TS1
totaliseSchema TS3
    
```

13.1.11 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 CS2 :: 
  "(*CS2_TYPES*) => bool"
where
"CS2 (*CS2_VARIABLES*) == (P02)"

lemma TS2:
"(*TS2_EXPRESSION*)"
sorry

definition CS1 :: 
  "(*CS1_TYPES*) => bool"
where
"CS1 (*CS1_VARIABLES*) == (P01)"

lemma TS1:
"(*TS1_EXPRESSION*)"
sorry

lemma TS3:
"(*TS3_EXPRESSION*)"
sorry

end
end

```

13.1.12 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_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 => 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
```

13.1.13 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 => bool"  
where  
  " VM_nosale stock takings stock' takings'  
  cash_refunded bars_delivered == ((stock' = stock)  
  ∧ (bars_delivered = 0)  
  ∧ (cash_refunded = cash_tendered)  
  ∧ (takings' = takings))"  
  
definition VM2 ::  
  "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 => 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 => 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:  
" (exists stock' takings' cash_refunded bars_delivered.  
VM1 cash_tendered stock takings stock' takings'  
cash_refunded bars_delivered)  
iff (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:  
"( $\exists$  stock' takings' cash_refunded bars_delivered.  
VM2 cash_tendered stock takings stock' takings'  
cash_refunded bars_delivered)  
 $\longleftrightarrow$  (cash_tendered < price)  $\wedge$   
(cash_tendered  $\geq$  0)  $\wedge$  (stock  $\geq$  0)  $\wedge$  (takings  $\geq$  0)"  
apply (unfold VM2_def insufficient_cash_def VM_nosale_def )  
apply auto  
done  
  
lemma pre_VM3:  
"( $\exists$  stock' takings' cash_refunded bars_delivered.  
VM3 cash_tendered stock takings stock'  
takings' cash_refunded bars_delivered)  
 $\longleftrightarrow$  (0 < stock  $\wedge$   
cash_tendered = price  $\wedge$  0  $\leq$  takings)  $\vee$  (cash_tendered < price)  
 $\wedge$  (0  $\leq$  cash_tendered)  
 $\wedge$  (0  $\leq$  stock)  
 $\wedge$  (0  $\leq$  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: " $\neg$  (insufficient_cash  
cash_tendered  $\wedge$  exact_cash cash_tendered)"  
apply (unfold insufficient_cash_def exact_cash_def)  
apply auto  
done
```

```
lemma VM3_refines_VM1:
  " $(\exists \text{stock}' \text{ takings}' \text{ cash_refunded} \text{ bars_delivered}.$ 
   ((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:
  " $(\exists \text{stock}' \text{ takings} \text{ cash_refunded} \text{ bars_delivered}.$ 
   (VM3 cash_tendered stock takings stock' takings' cash_refunded
   bars_delivered)
  →
   ((\text{takings}' - \text{takings}) \geq \text{price} * (\text{stock} - \text{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
```

13.2 BirthdayBook

13.2.1 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}
```

13.2.2 Raw Latex output

$[NAME, DATE]$

<i>BirthdayBook</i>	_____
$\text{known} : \mathbb{P} NAME$	
$\text{birthday} : NAME \rightarrow DATE$	
$\text{known} = \text{dom } \text{birthday}$	

<i>InitBirthdayBook</i>	_____
$\text{BirthdayBook}'$	
$\text{known}' = \{\}$	

<i>AddBirthday</i>	_____
$\Delta \text{BirthdayBook}$	
$\text{name?} : NAME$	
$\text{date?} : DATE$	
$\text{name?} \notin \text{known}$	
$\text{birthday}' = \text{birthday} \cup \{\text{name?} \mapsto \text{date?}\}$	

<i>FindBirthday</i>	_____
$\exists \text{BirthdayBook}$	
$\text{name?} : NAME$	
$\text{date!} : DATE$	
$\text{name?} \in \text{known}$	
$\text{date!} = \text{birthday}(\text{name?})$	

$REPORT ::= ok \mid already_known \mid not_known$

<i>Success</i>	_____
$\text{result!} : REPORT$	
$\text{result!} = ok$	

<i>NotKnown</i>	_____
$\exists \text{BirthdayBook}$	
$\text{name?} : NAME$	
$\text{result!} : REPORT$	
$\text{name?} \notin \text{known}$	
$\text{result!} = not_known$	

<i>AlreadyKnown</i>	_____
$\exists \text{BirthdayBook}$	
$\text{name?} : NAME$	
$\text{result!} : REPORT$	
$\text{name?} \in \text{known}$	
$\text{result!} = already_known$	

$RAddBirthday ==$
 $(AddBirthday \wedge Success)$
 \vee *AlreadyKnown*
 $RFindBirthday ==$
 $(FindBirthday \wedge Success) \vee NotKnown$

13.2.3 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{\{} \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}}} \\
\text{\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}}} \\
\text{\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}}} \\
\text{\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}}} \\
\text{\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}

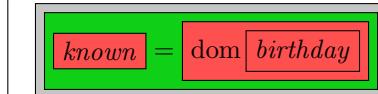
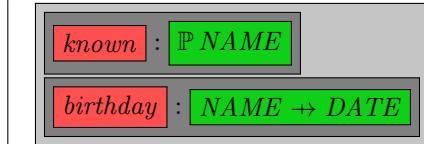
```

13.2.4 ZCGa output

\boxed{NAME}

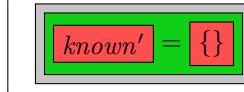
\boxed{DATE}

BirthdayBook _____



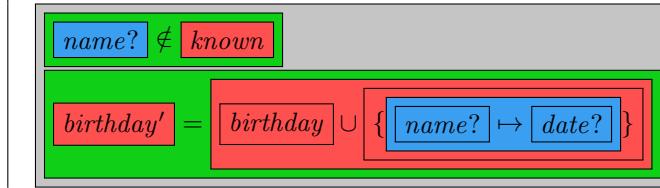
InitBirthdayBook _____

$\boxed{BirthdayBook}$

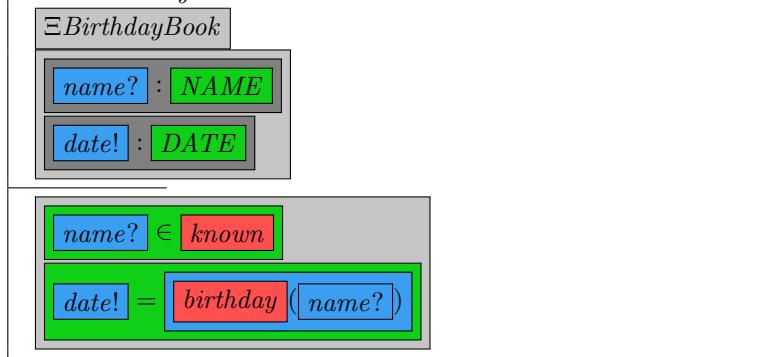


AddBirthday _____

$\Delta BirthdayBook$



FindBirthday

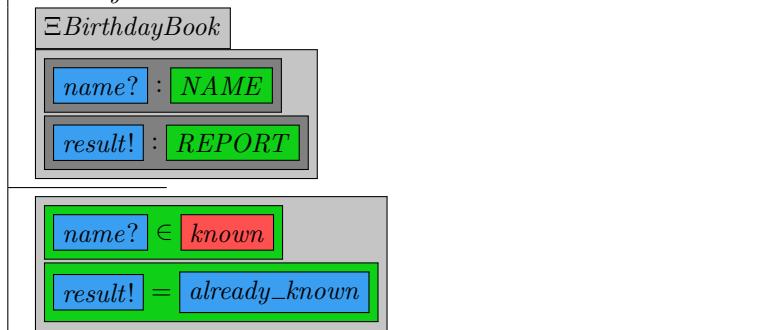


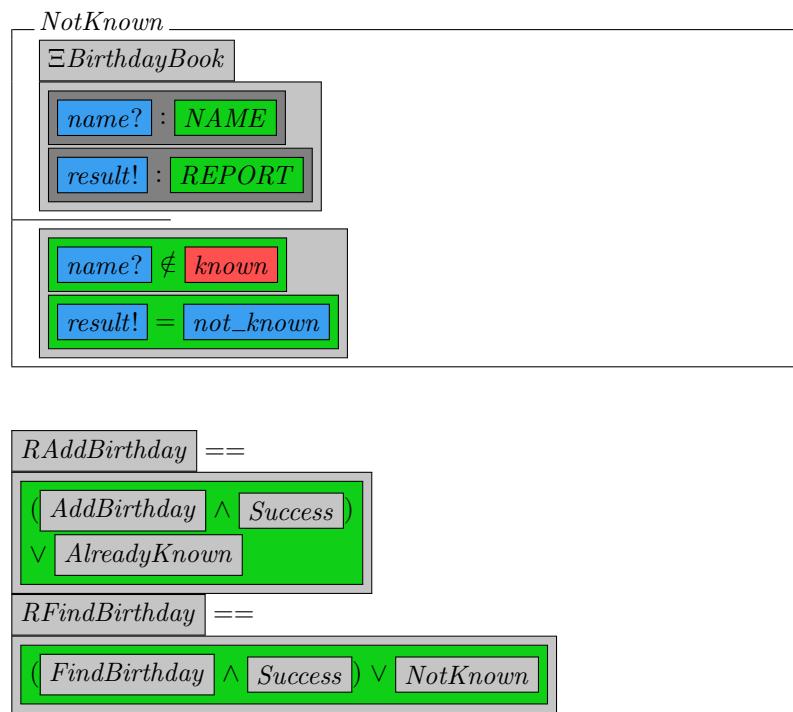
$\text{REPORT} ::= ok \mid already_known \mid not_known$

Success



AlreadyKnown





13.2.5 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{P02}{known' = \{} \}}
\end{schema}
}

\requires{IS1}{P02}
\initialof{IS1}{SS1}

\draschema{CS1}{
\begin{schema}{AddBirthday}
\Delta BirthdayBook \\
name?: NAME \\
date?: DATE
\where
\draline{PRE1}{name? \notin known} \\
\draline{P03}{birthday' = birthday \cup \\
\{name? \mapsto date?\}}
\end{schema}
}

\uses{CS1}{IS1}
\requires{CS1}{PRE1}
\allows{PRE1}{P03}

\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}
}

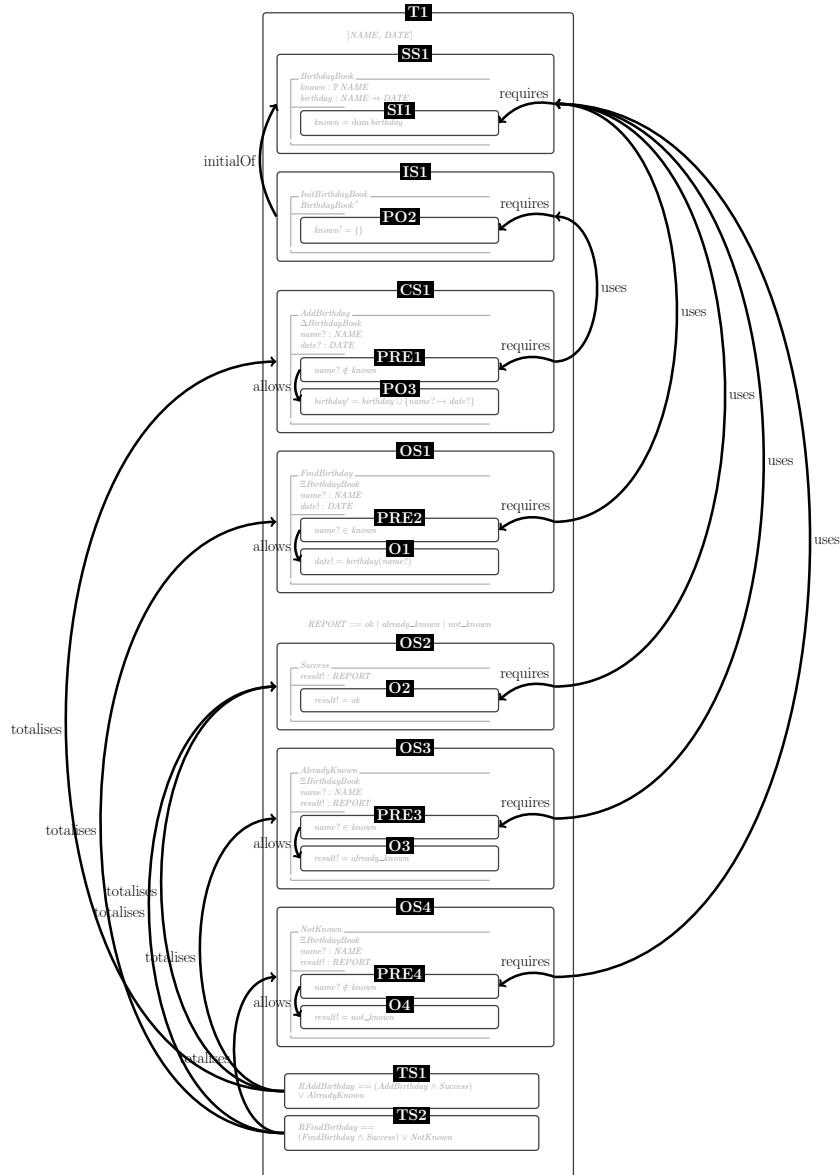
\requires{OS4}{PRE4}
\allows{PRE4}{O4}
\uses{OS4}{SS1}

\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}
}

```

13.2.6 ZDRa Output



13.2.7 ZCGa and ZDRA Annotated Latex Code

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

\begin{document}

\drattheory{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{P02}{\text{\expression{\set{known}' } = \set{\set{}}}}
\end{schema}
}

\requires{IS1}{P02}

\initialof{IS1}{SS1}

\draschema{CS1}{
\begin{schema}{AddBirthday}
\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{P03}{\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}{P03}
```

```
\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!}} = }} \\
\text{\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}

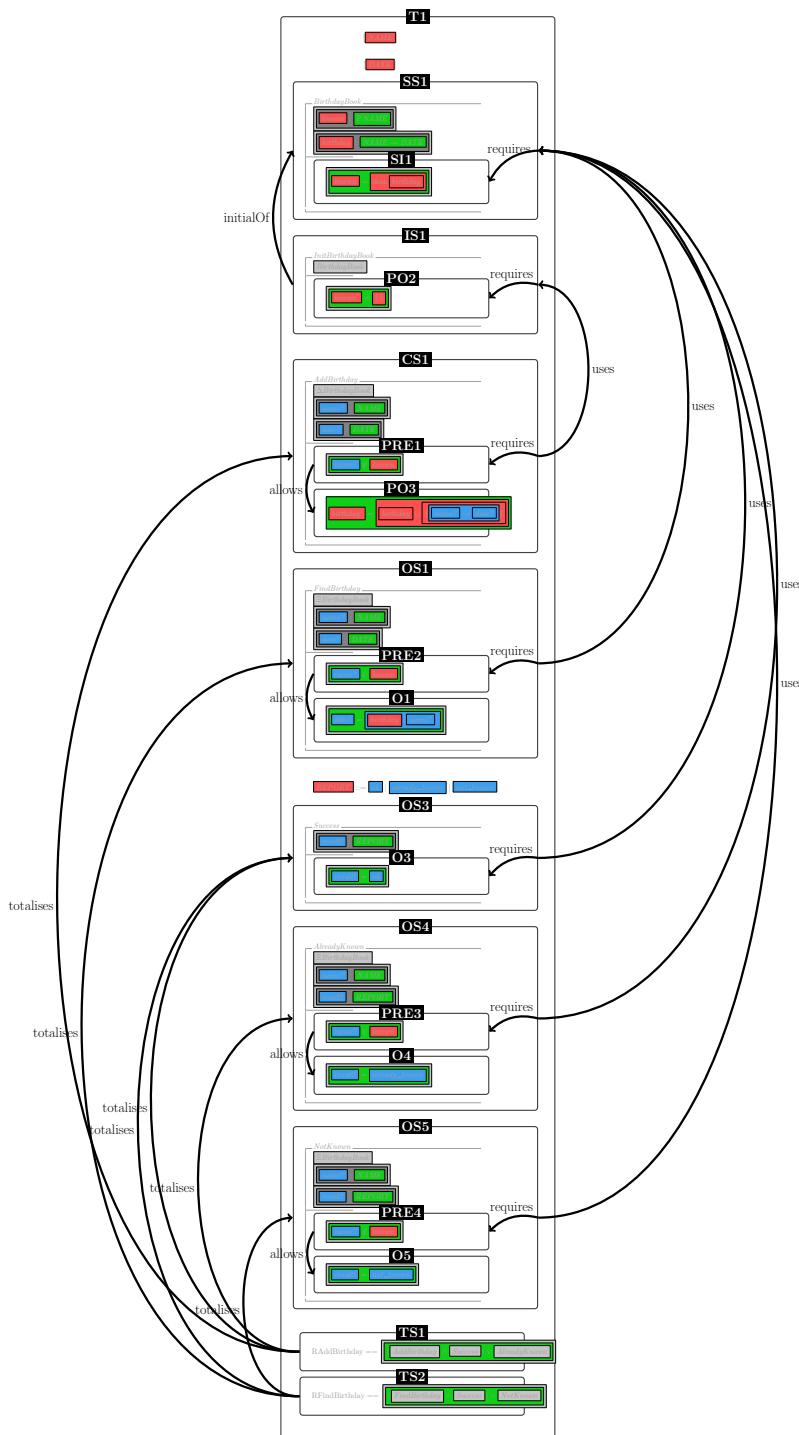
\requires{OS5}{PRE4}
\allows{PRE4}{O5}
\uses{OS5}{SS1}

\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}

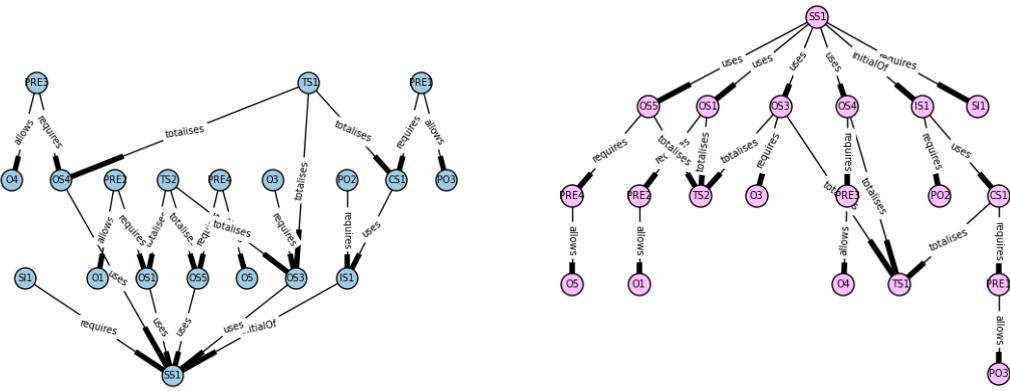
\begin{aligned}
&\totalises{TS1}{CS1} \\
&\totalises{TS1}{OS3} \\
&\totalises{TS1}{OS4} \\
&\totalises{TS2}{OS1} \\
&\totalises{TS2}{OS3} \\
&\totalises{TS2}{OS5}
\end{aligned}
}

\end{document}
```

13.2.8 ZCGa and ZDRA output



13.2.9 Dependency and Goto Graphs



13.2.10 General Proof Skeleton

stateSchema SS1	precondition PRE3
stateInvariants SI1	postcondition P03
initialSchema IS1	output 05
postcondition P02	output 04
outputSchema OS1	outputSchema OS3
precondition PRE2	output 03
changeSchema CS1	output 01
precondition PRE1	totaliseSchema TS2
outputSchema OS5	totaliseSchema TS1
precondition PRE4	lemma L1_(CS1)
outputSchema OS4	

13.2.11 Isabelle Proof Skeleton

```

theory 4
imports
Main

begin

record SS1 =
(*DECLARATIONS*)

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

definition IS1 :: 
"(*IS1_TYPES*) => bool"
where
"IS1 (*IS1_VARIABLES*) == (P02)"

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

definition CS1 :: 
"(*CS1_TYPES*) => bool"
where
"CS1 (*CS1_VARIABLES*) ==
(PRE1)
\wedge (P03)"

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

definition OS4 :: 
"(*OS4_TYPES*) => bool"
where
"OS4 (*OS4_VARIABLES*) == (PRE3)
\wedge (O4)"

definition OS3 :: 
"(*OS3_TYPES*) => bool"
where
"OS3 (*OS3_VARIABLES*) == (O3)"

definition OS2 :: 
"(*OS2_TYPES*) => bool"
where
"OS3 (*OS2_VARIABLES*) == (O2)"

definition TS2 :: 
"(*TS2_TYPES*) => bool"
where
"OS3 (*TS2_VARIABLES*) ==
(*TS2_EXPRESSION*)"

definition TS1 :: 
"(*TS1_TYPES*) => bool"
where
"OS3 (*TS1_VARIABLES*) ==
(*TS1_EXPRESSION*)"

end
end

```

13.2.12 Isabelle Filled In

```

theory new5
imports
Main

begin
typedcl NAME
typedcl 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

```

13.2.13 Full Proof in Isabelle

```

theory new6
imports
Main

begin
typedef NAME
typedef 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')
 $\leftrightarrow$ 
(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)
 $\rightarrow$  known = dom birthday"
apply (rule impI)
apply (unfold birthdaybookstate_def)
apply auto
done

lemma InitisOk:
"(∃ birthdaybook. InitBirthdayBook birthdaybook' known' birthday' birthdaybook)
 $\leftrightarrow$  (known' = {}) "
apply (unfold InitBirthdayBook_def)
apply auto
done

lemma RAddBirthdayIsTotal:
"(∃ known' birthday' birthdaybook
birthdaybook' date.
RAddBirthday known' name birthdaybook birthdaybook' date birthday' result)
 $\rightarrow$ 
(name ∉ known) ∨ (name ∈ known)"
apply (unfold RAddBirthday_def)
apply (unfold AddBirthday_def AlreadyKnown_def Success_def)
apply auto
done
end
end


```

13.3 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 [?].

13.3.1 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 \set{phoneNumbers} \\
phoneNumbers' = phoneNumbers \cup
\{n \mapsto p\}
\end{schema}

\begin{schema}{RemovePerson}
\Delta TelephoneDirectory \\
n?: NAME \\
s?: SURNAME \\
p?: TELEPHONE
\where
n? \mapsto s? \in persons\\
n? \mapsto p? \notin \set{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 \set{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}

```

13.3.2 Raw Latex output

[NAME]

[SURNAME]

<i>TelephoneDirectory</i>	_____
<i>persons</i> : NAME → SURNAME	
<i>phoneNumbers</i> : NAME ↗ TELEPHONE	
<i>dom phoneNumbers</i> = <i>dom persons</i>	

<i>InitTelephoneDirectory</i>	_____
<i>TelephoneDirectory'</i>	
<i>phoneNumbers'</i> = {}	
<i>persons'</i> = {}	

<i>AddPerson</i>	_____
<i>TheTelephoneDirectory</i>	
<i>name?</i> : NAME	
<i>surname?</i> : SURNAME	
<i>name? ↗ surname? ∈ persons</i>	
<i>persons'</i> = <i>persons</i> ∪ { <i>name? ↗ surname?</i> }	

<i>AddNumber</i>	_____
Δ <i>TelephoneDirectory</i>	
<i>n?</i> : NAME	
<i>s?</i> : SURNAME	
<i>p?</i> : TELEPHONE	
<i>n ↗ s ∈ persons</i>	
<i>p? ∉ ran phoneNumbers</i>	
<i>phoneNumbers'</i> = <i>phoneNumbers</i> ∪ { <i>n ↗ phone?</i> }	

$OUTPUT ::= success \mid alreadyInDirectory \mid nameNotInDirectory \mid numberInUse$

<i>Success</i>	_____
message! : $OUTPUT$	
message! = <i>success</i>	

<i>AlreadyInDirectory</i>	_____
message! : $OUTPUT$	
n? : NAME	
s? : SURNAME	
n? \mapsto s \in person	
message! = <i>alreadyInDirectory</i>	

<i>NameNotInDirectory</i>	_____
message! : $OUTPUT$	
n? : NAME	
s? : SURNAME	
n? \mapsto s? \notin persons	
message! = <i>NotInDirectory</i>	

<i>NumberInUse</i>	_____
message! : $OUTPUT$	
p? : TELEPHONE	
p? \in ran phoneNumbers	
message! = <i>numberInUse</i>	

$TotalAddPerson \hat{=} (AddPerson \wedge Success) \vee AlreadyInDirectory$

$TotalRemovePerson \hat{=} (RemovePerson \wedge Success) \vee NameNotInDirectory$

$TotalAddNumber \hat{=} (AddNumber \wedge Success) \vee NameNotInDirectory \vee NumberInUse$

13.3.3 ZCGa and ZDRa Annotated Latex Code

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

\begin{document}

\drattheory{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{P01}{} \\
\text{\expression{\set{phoneNumbers'} = \set{\{\}}}} \\
\text{\expression{\set{persons'} = \set{\{\}}}} \\
\end{schema}
}

\initialof{IS1}{SS1}
\requires{IS1}{P01}

\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{\mapsto} \term{surname?} \notin \set{persons}}}} \\
\draline{P02}{} \\
\text{\expression{\set{persons'} = \set{\set{persons} \cup \set{\term{\mapsto} \term{surname?}}}}}} \\
\end{schema}
}

\uses{CS1}{IS1}
\requires{CS1}{PRE1}
\allows{PRE1}{P02}

\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{\mapsto} \term{s?} \in \set{persons}}}} \\
\text{\expression{\term{p?} \notin \set{\ran \set{phoneNumbers}}}} \\
\draline{P03}{} \\
\text{\expression{\set{phoneNumbers'} = \set{\set{phoneNumbers} \cup \set{\term{\mapsto} \term{phone?}}}}}} \\
\end{schema}
}

\uses{CS2}{IS1}
\allows{PRE2}{P03}
\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{n?} \mapsto \term{s?} \in \set{persons}}} \\
\text{\expression{\term{n?} \mapsto \term{p?} \notin \set{phoneNumbers}}}} \\
\draline{P04}{

\text{\set{\set{persons'}} = \set{\set{persons} \setminus \set{\term{n?} \mapsto \term{s?}}}} \\
\end{schema}

\uses{CS3}{IS1}
\allows{PRE3}{P04}
\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{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{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}

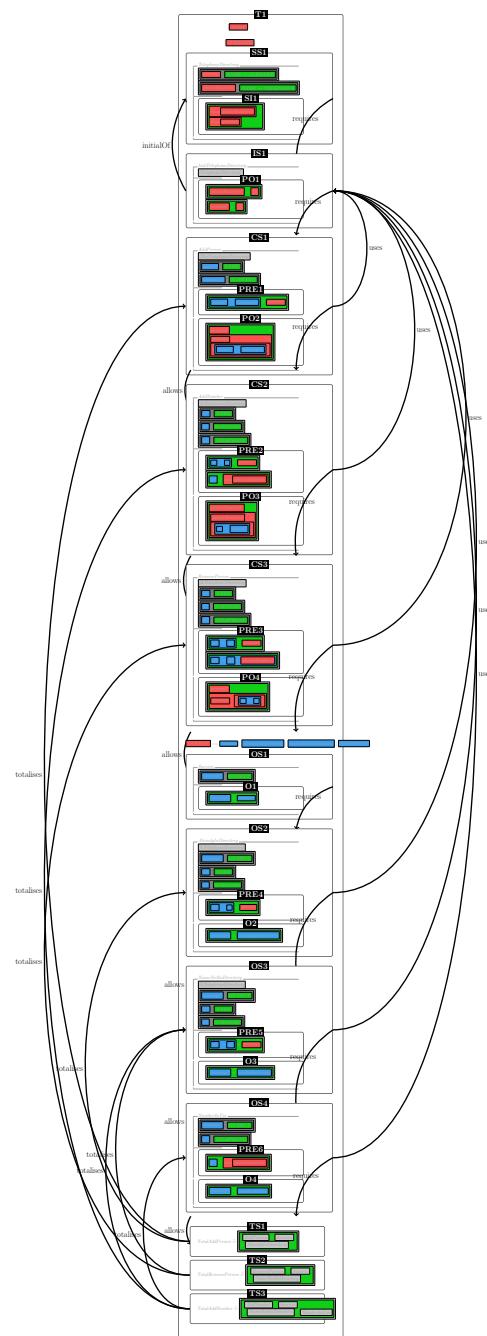
\totalises{TS1}{CS1}
\totalises{TS1}{OS2}
\totalises{TS2}{CS3}
\totalises{TS2}{OS3}
\totalises{TS3}{CS2}
\totalises{TS3}{OS4}
\totalises{TS3}{OS3}

}

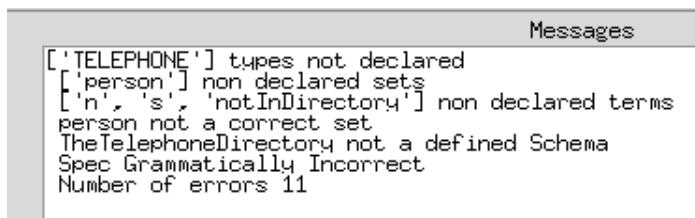
\end{document}

```

13.3.4 ZCGa and ZDRA output



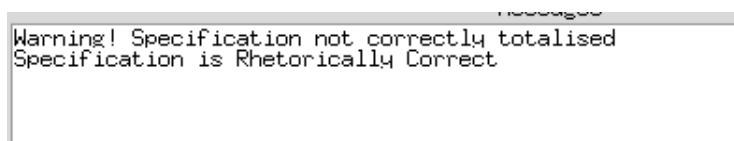
13.3.5 Messages when running the specification through the ZCGa and ZDRa checks



The screenshot shows a terminal window with a title bar labeled "Messages". The main area contains the following text:

```
[ 'TELEPHONE' ] types not declared
[ 'person' ] non declared sets
[ 'n', 's', 'notInDirectory' ] non declared terms
person not a correct set
TheTelephoneDirectory not a defined Schema
Spec Grammatically Incorrect
Number of errors 11
```

Figure 13.1: Message when checking the specification for ZCGa correctness.



The screenshot shows a terminal window with a title bar labeled "Messages". The main area contains the following text:

```
Warning! Specification not correctly totalised
Specification is Rhetorically Correct
```

Figure 13.2: Message when checking the specification for ZDRa correctness.

13.4 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 [?].

13.4.1 Raw Latex output

[NAME]

[SURNAME]

[TELEPHONE]

TelephoneDirectory

$\text{persons} : \text{NAME} \rightarrow \text{SURNAME}$

$\text{phoneNumbers} : \text{NAME} \rightarrow \text{TELEPHONE}$

$\text{dom phoneNumbers} = \text{dom persons}$

InitTelephoneDirectory

TelephoneDirectory'

$\text{phoneNumbers}' = \{\}$

$\text{persons}' = \{\}$

AddPerson

$\Delta \text{TelephoneDirectory}$

$\text{name?} : \text{NAME}$

$\text{surname?} : \text{SURNAME}$

$\text{phone?} : \text{TELEPHONE}$

$\text{name?} \mapsto \text{surname?} \notin \text{persons}$

$\text{persons}' = \text{persons} \cup \{\text{name?} \mapsto \text{surname?}\}$

AddNumber

$\Delta \text{TelephoneDirectory}$

$\text{n?} : \text{NAME}$

$\text{s?} : \text{SURNAME}$

$\text{p?} : \text{TELEPHONE}$

$\text{n?} \mapsto \text{s?} \in \text{persons}$

$\text{p?} \notin \text{ran phoneNumbers}$

$\text{phoneNumbers}' = \text{phoneNumbers} \cup \{\text{name?} \mapsto \text{phone?}\}$

RemovePerson

$\Delta \text{TelephoneDirectory}$

$\text{n?} : \text{NAME}$

$\text{s?} : \text{SURNAME}$

$\text{p?} : \text{TELEPHONE}$

$\text{n?} \mapsto \text{s?} \in \text{persons}$

$\text{n?} \mapsto \text{p?} \notin \text{phoneNumbers}$

$\text{persons}' = \text{persons} \setminus \{\text{n?} \mapsto \text{s?}\}$

RemoveNumber _____

$\Delta \text{TelephoneDirectory}$
 $p? : \text{TELEPHONE}$

$p? \in \text{ran } \text{phoneNumbers}$

$\exists m : \text{dom } \text{persons} \bullet$

$m \mapsto p? \in \text{phoneNumbers} \wedge$

$\text{phoneNumbers}' = \text{phoneNumbers} \setminus \{m \mapsto p?\}$

OUTPUT ::= success | alreadyInDirectory | nameNotInDirectory | numberInUse | numberDoesntExist

Success _____

$\text{message!} : \text{OUTPUT}$

$\text{message!} = \text{success}$

AlreadyInDirectory _____

$\exists \text{TelephoneDirectory}$

$\text{message!} : \text{OUTPUT}$

$n? : \text{NAME}$

$s? : \text{SURNAME}$

$n? \mapsto s? \in \text{persons}$

$\text{message!} = \text{alreadyInDirectory}$

NameNotInDirectory _____

$\exists \text{TelephoneDirectory}$

$\text{message!} : \text{OUTPUT}$

$n? : \text{NAME}$

$s? : \text{SURNAME}$

$n? \mapsto s? \notin \text{persons}$

$\text{message!} = \text{nameNotInDirectory}$

NumberInUse _____

$\text{message!} : \text{OUTPUT}$

$p? : \text{TELEPHONE}$

$p? \in \text{ran } \text{phoneNumbers}$

$\text{message!} = \text{numberInUse}$

<i>NumberDoesntExist</i>	_____
message! : OUTPUT	
p? : TELEPHONE	
p? \notin ran phoneNumbers	
message! = numberDoesntExist	

TotalAddPerson $\hat{=}$ (*AddPerson* \wedge *Success*) \vee *AlreadyInDirectory*

TotalRemovePerson $\hat{=}$ (*RemovePerson* \wedge *Success*)

\vee *NameNotInDirectory*

TotalAddNumber $\hat{=}$ (*AddNumber* \wedge *Success*)

\vee *NameNotInDirectory* \vee *NumberInUse*

TotalRemoveNumber $\hat{=}$ (*RemoveNumber* \wedge *Success*)

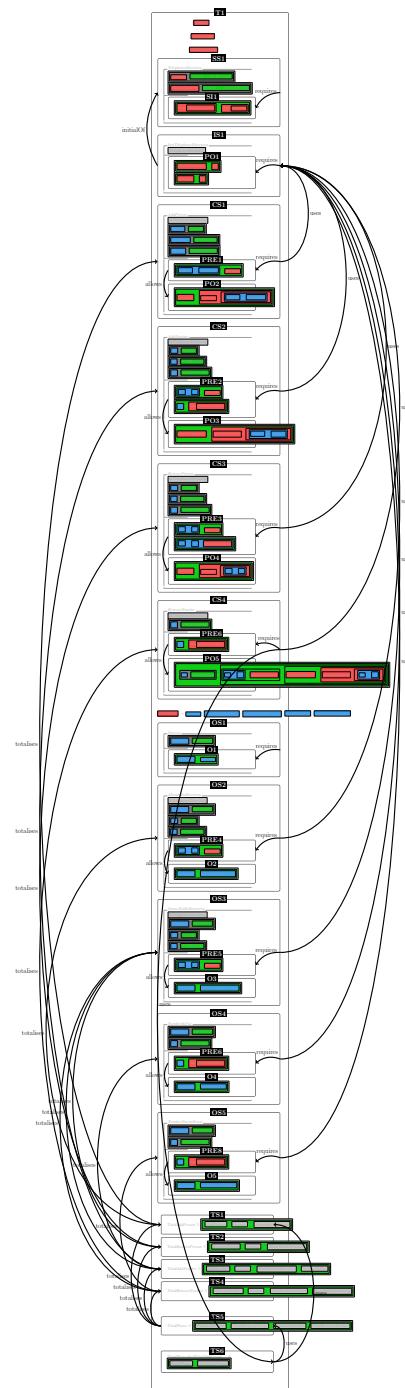
\vee *NumberDoesntExist* \vee *NameNotInDirectory*

TotalPhone $\hat{=}$ *TotalAddPerson* \vee *TotalRemovePerson*

\vee *TotalAddNumber* \vee *TotalRemoveNumber*

TotalPhoneAndTotalAddPerson $\hat{=}$ *TotalPhone* \vee *TotalAddPerson*

13.4.2 ZCGa and ZDRA output



13.4.3 Messages when running the specification through the ZCGa and ZDRA checks

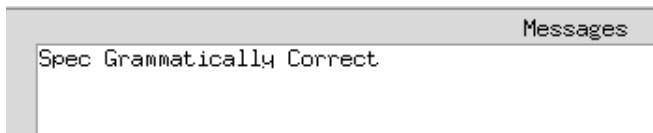


Figure 13.3: Message when checking the specification for ZCGa correctness.

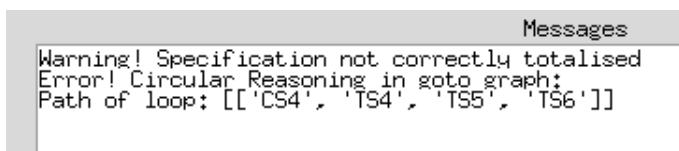


Figure 13.4: Message when checking the specification for ZDRA correctness.

13.5 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 language specification which is on its way to being formalised.

13.5.1 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 | press_cas_eng | press_alt_eng |
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 | engaged
```

<i>off_eng</i>	_____
<i>mode : mode_status</i>	
<i>mode = off</i> \vee <i>mode = engaged</i>	

<i>AutoPilot</i>	_____
<i>att_cws : mode_status</i>	
<i>fpa_sel : mode_status</i>	
<i>alt_eng : mode_status</i>	
<i>cas_eng : mode_status</i>	

<i>att_cwsDo</i>	_____
Δ <i>AutoPilot</i>	
<i>att_cws = off</i>	
<i>att_cws' = engaged</i>	
<i>fpa_sel' = off</i>	
<i>alt_eng' = off</i>	
<i>cas_eng' = off</i> \vee <i>engaged</i>	

2. There are three displays on the panel: 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 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 modeled.

13.5.2 ZCGa and ZDRa Annotated Latex Code

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

\begin{document}

\drathteo{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}
\end{enumerate}

\begin{zed}
\set{events} ::= \term{press\_\textit{att\_cws}} | \term{press\_\textit{cas\_eng}} | \term{press\_\textit{alt\_eng}} | \\
\term{press\_\textit{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 alt\_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}

\begin{draschema}{OS1}{}
\begin{schema}{off\_eng}
\text{\declaration{\term{mode}: \expression{mode\_status}}}
\where
\begin{draschema}{O1}{}
\text{\expression{\expression{\term{mode} = \term{off}} \lor \expression{\term{mode} = \term{engaged}}}}
\end{schema}
\end{draschema}
\requires{OS1}{O1}

\begin{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}
\end{draschema}
\end{zed}
```

```
\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{P01} {
\text{\expression{\term{att\_cws}} = \term{engaged}}} \\
\text{\expression{\term{fpa\_sel}} = \term{off}}} \\
\text{\expression{\term{alt\_eng}} = \term{off}}} \\
\text{\expression{\term{cas\_eng}} = \term{off}}} \text{\lor} \\
\text{\expression{\term{cas\_eng}} = \term{engaged}}}
\end{schema}

\uses{CS1}{SS2}
\allows{PRE1}{P01}
\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 alt\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.

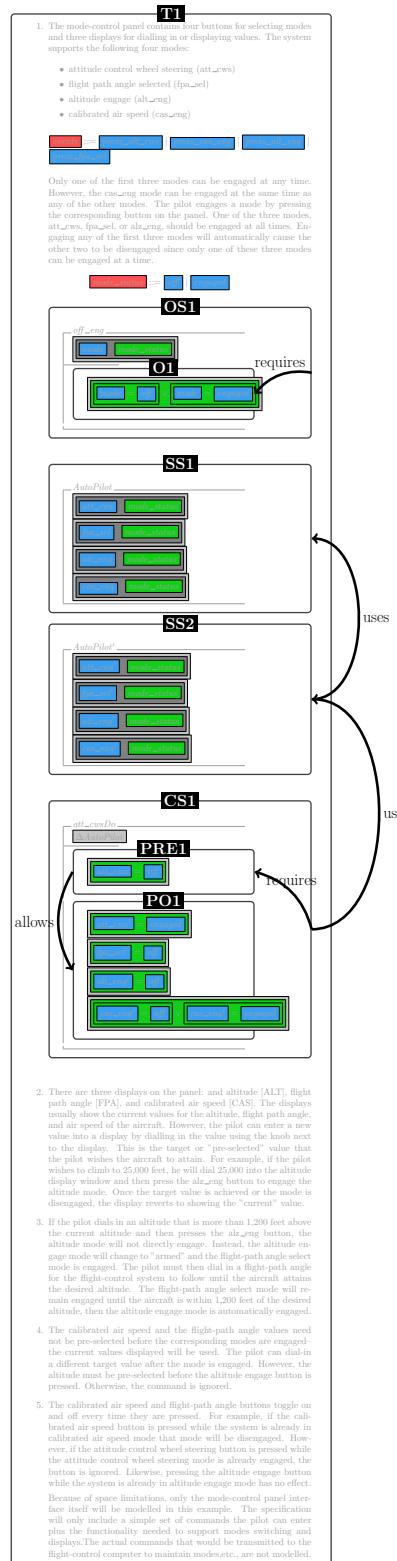
\item If the pilot dials in an altitude that is more than 1,200 feet above the current altitude and then presses the alt\eng button, the altitude mode will not directly engage. Instead, the altitude engage mode will change to "armed" and the flight-path angle select mode is engaged. The pilot must then 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}
```

13.5.3 ZCGa and ZDRA output



13.5.4 Messages when running the specification through the ZCGa and ZDRa checks

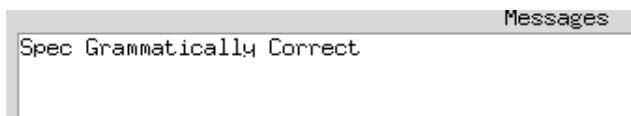


Figure 13.5: Message when checking the specification for ZCGa correctness.

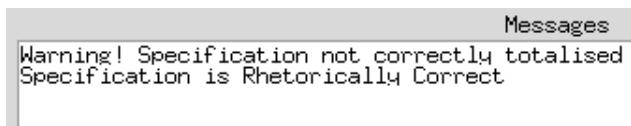


Figure 13.6: Message when checking the specification for ZDRa correctness.

13.5.5 General Proof Skeleton

```
stateSchema SS1
    outputSchema OS1
    output O1
stateSchema SS2
changeSchema CS1
precondition PRE1
postcondition PO1
```

13.5.6 Isabelle Proof Skeleton

```
theory gpsaln2
imports
Main

begin
(*DATATYPES*)

record SS1 =
(*DECLARATIONS*)

locale ln2 =
fixes (*GLOBAL DECLARATIONS*)
begin

definition OS1 :: 
"(*OS1_TYPES*) => bool"
where
"OS1 (*OS1_VARIABLES*) == (01)"

definition CS1 :: 
"(*CS1_TYPES*) => bool"
where
"CS1 (*CS1_VARIABLES*) ==
(PRE1)
^ (P01)"

end
end
```

13.5.7 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)"
```

13.6 ModuleReg

13.6.1 ModuleReg Full Proof

This section shows the full proof for the modulereg example from [?]. It includes the filled in Isabelle skeleton. Along with added proofs which have been input by

the user.

```
theory new6
imports
Main

begin
typedcl PERSON
typedcl 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 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')
 $\longleftrightarrow$  (p ∈ students)
 $\wedge$  (m ∈ degModules)
 $\wedge$  ((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' = {})
 $\wedge$  (degmodules' = {})
 $\wedge$  (taking' = {}))"

lemma InitOK:
"(∃ modulereg'. InitModuleReg modulereg' students' degmodules' taking')
 $\longrightarrow$  ((

(students' = {})
 $\wedge$  (degmodules' = {})
 $\wedge$  (taking' = {}))
 $\wedge$  ((Domain taking' ⊆ students')
 $\wedge$  (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)
 $\longrightarrow$  (students' ≠ {})
 $\wedge$  (degModules' ≠ {})
by (smt RegForModule_def empty_iff empty_iff)

lemma notEmpty:
"(taking' = taking ∪ {(p, m)})  $\longrightarrow$  (taking' ≠ {})"
by (smt Un_empty insert_not_empty)

end
end

```

13.7 SteamBoiler

13.7.1 Raw Latex output

State ::= *init* | *norm* | *broken* | *stop*

OnOff ::= *on* | *off*

OpenClosed ::= *open* | *closed*

Physical Constants

$w_{min} : \mathbb{N}$

$w_{max} : \mathbb{N}$

$l : \mathbb{N}$

$d_{max} : \mathbb{N}$

$\delta_p : \mathbb{N}$

$\delta_d : \mathbb{N}$

$w_{min} < w_{max}$

Measured values

Input

$w? : \mathbb{N}$

$d? : \mathbb{N}$

Control values

Pumps

$p_1, p_2, p_3, p_4 : OnOff$

SteamBoiler0

Pumps

$v : OpenClosed$

$a : OnOff$

$z : State$

Auxiliary Schemata

PumpsOff

Pumps'

$p'_1 = off \wedge p'_2 = off \wedge p'_3 = off \wedge p'_4 = off$

PumpsOn

Pumps'

$p'_1 = on \wedge p'_2 = on \wedge p'_3 = on \wedge p'_4 = on$

Steam Boiler Initial State

<i>SteamBoilerInit0</i>	_____
<i>SteamBoiler0'</i>	_____
<i>a' = off</i>	_____
<i>z' = init</i>	_____

Operations for Initialisation

<i>SInitNormal0</i>	_____
<i>ΔSteamBoiler0</i>	_____
<i>Input</i>	_____
<i>z = init</i>	_____
<i>d? = 0</i>	_____
<i>w? ≥ w_{min} + d_{max}</i>	_____
<i>w? ≤ w_{max}</i>	_____
<i>PumpsOff</i>	_____
<i>z' = norm</i>	_____
<i>v' = closed</i>	_____
<i>a' = on</i>	_____

<i>SInitStop0</i>	_____
<i>ΔSteamBoiler0</i>	_____
<i>Input</i>	_____
<i>z = init</i>	_____
<i>d? > 0</i>	_____
<i>z' = stop</i>	_____

<i>SInitFill0</i>	_____
<i>ΔSteamBoiler0</i>	_____
<i>Input</i>	_____
<i>z = init</i>	_____
<i>d? = 0</i>	_____
<i>w? < w_{min} + d_{max}</i>	_____
<i>PumpsOn</i>	_____
<i>z' = z</i>	_____
<i>v' = closed</i>	_____
<i>a' = off</i>	_____

$SInitEmpty0$

$\Delta_{SteamBoiler0}$

Input

$z = init$

$d? = 0$

$w? > w_{max}$

$PumpsOff$

$z' = z$

$v' = open$

$a' = off$

$ControlInit0 \hat{=} SInitNormal0$

$\vee SInitStop0$

$\vee SInitFill0$

$\vee SInitEmpty0$

Operations for Normal State

$SNormalFill0$

$\Delta_{SteamBoiler0}$

Input

$z = norm$

$w?? \geq w_{min}$

$w? \leq w_{opt} - 3l$

$PumpsOn$

$v' = closed \wedge a' = on \wedge z' = z$

Note: Simplified version where all four pumps are switched on simultaneously.

$SNormalContinue0$

$\Xi_{SteamBoiler0}$

Input

$z = norm$

$w? > w_{opt} - 3l$

$w? \leq w_{opt}$

$SNormalNotFill0$ —————

$\Delta_{SteamBoiler0}$

Input

$z = norm$

$w? > w_{opt}$

$w? \leq w_{max}$

$PumpsOff$

$v' = closed \wedge a' = on \wedge z' = z$

$SNormalStop0$ —————

$\Delta_{SteamBoiler0}$

Input

$z = norm$

$w? < w_{min} \vee w? > w_{max}$

$a' = off \wedge z' = stop$

$ControlNormal0 \hat{=} SNormalFill0$

$\vee SNormalContinue0$

$\vee SNormalNotFill0$

$\vee SNormalStop0$

$Control0 \hat{=} ControlInit0$

$\vee ControlNormal0$

Extended Solution

Additional Type

$WorksBroken ::= works \mid broken$

Additional measured values

$ControlInput$ —————

$k_w? : WorksBroken$

$k_d? : WorksBroken$

$k_{p1}? : WorksBroken$

$k_{p2}? : WorksBroken$

$k_{p3}? : WorksBroken$

$k_{p4}? : WorksBroken$

Control values

$SteamBoiler1$ —————

$SteamBoiler0$

$s : \mathbb{N}$

$\delta : \mathbb{N}$

Initial State

<i>SteamBoilerInit1</i>	_____
<i>SteamBoiler1'</i>	_____
$a' = off$	_____
$z' = init$	_____

Auxiliary Functions

$pswitch : (OnOff \times WorksBroken) \rightarrow OnOff$	_____
$pswitch(on, works) = on$	_____
$pswitch(on, broken) = off$	_____
$pswitch(off, works) = off$	_____
$pswitch(off, broken) = off$	_____

$pamount : (OnOff \times WorksBroken) \rightarrow \mathbb{N}$	_____
$\forall x : OnOff, y : WorksBroken \mid x = off \vee y = broken \bullet$	_____
$pamount(x, y) = 0$	_____
$pamount(on, works) = 1$	_____

Auxiliary Schemata

<i>Pumps ControlledOn</i>	_____
<i>Pumps'</i>	_____
<i>ControllInput</i>	_____
$p'_1 = pswitch(on, k_{p1}?) \wedge p'_2 = pswitch(on, k_{p2}?)$	_____
$p'_2 = pswitch(on, k_{p3}?) \wedge p'_4 = pswitch(on, k_{p4}?)$	_____

<i>Pumps ControlledOff</i>	_____
<i>Pumps'</i>	_____
<i>ControllInput</i>	_____
$p'_1 = pswitch(off, k_{p1}?) \wedge p'_2 = pswitch(off, k_{p2}?)$	_____
$p'_2 = pswitch(off, k_{p3}?) \wedge p'_4 = pswitch(off, k_{p4}?)$	_____

Operations for Initialisation

SInitNormal1

$\Delta_{SteamBoiler1}$

Input

ControlInput

$z = init$

$d? = 0$

$k_w = works \wedge k_d = works$

$w? \geq w_{min} + d_{max}$

$w? \leq w_{max}$

$z' = norm$

$v' = closed$

$a' = on$

$s' = w?$

PumpsOff

SInitFill1

$\Delta_{SteamBoiler1}$

Input

ControlInput

$z = init$

$d? = 0 k_w = works \wedge k_d = works$

$w? < w_{min} + d_{max}$

$z' = z$

$v' = closed$

$a' = off$

PumpsOn

SInitEmpty1

$\Delta_{SteamBoiler1}$

Input

ControlInput

$z = init$

$d? = 0$

$w? > w_{max}$

$z' = z$

$v' = open$

$a' = off$

PumpsOff

$SInitStop1$

$\Delta_{SteamBoiler1}$

Input

ControlInput

$z = init$

$d? > 0 \vee K_w = broken \vee k_d = broken$

$z' = stop$

$ControlInit1 \hat{=} SInitNormal1$

$\vee SInitFill1$

$\vee SInitEmpty1$

$\vee SInitStop1$

Operations for Normal State

$SNormalFill1$

$\Delta_{SteamBoiler1}$

Input

ControlInput

$z = norm$

$k_w = works$

$w? \geq w_{min}$

$w? \leq w_{opt} = 3l$

$s' = w?$

Pumps Controlled On

$v' = closed \wedge a' = on \wedge z' = z$

$SNormalContinue1$

$\Delta_{SteamBoiler1}$

Input

ControlInput

$z = norm$

$k_w = works$

$w? > w_{opt} - 3l$

$w? \leq w_{opt}$

$p'_1 = pswitch(p_1, k_{p1}) \wedge p'_2 = pswitch(p_2, k_{p2})$

$p'_3 = pswitch(p_3, k_{p3}) \wedge p'_4 = pswitch(p_4, k_{p4})$

$s' = w?$

$v' = v \wedge a' = a \wedge z' = z$

<i>SNormalNotFill1</i>	_____
$\Delta_{SteamBoiler1}$	
<i>Input</i>	
<i>ControlInput</i>	
$z = norm$	
$k_w = works$	
$w? > w_{opt}$	
$w? \leq w_{max}$	
$s' = w?$	
<i>Pumps ControlledOff</i>	
$v' = closed \wedge a' = on \wedge z' = z$	

<i>SNormalWaterStop1</i>	_____
$\Delta_{SteamBoiler}$	
<i>Input</i>	
<i>ControlInput</i>	
$z = norm \vee z = broken$	
$k_w = works$	
$w? < w_{min} \vee w? > w_{max}$	
$a' = off \wedge z' = stop$	

<i>SNormalControlStop1</i>	_____
$\Delta_{SteamBoiler1}$	
<i>Input</i>	
<i>ControlInput</i>	
$z = norm$	
$k_w = broken \wedge k_d = broken$	
$a' = off \wedge z' = stop$	

<i>AmountComputation</i>	_____
$SteamBoiler1$	
<i>ControlInput</i>	
$amount : \mathbb{N}$	
$\delta_{pumps} : \mathbb{N}$	
$amount = l * (pamount(p_1, k_{p1}?)) + pamount(p_2, k_{p2}?)) +$	
$pamount(p_3, k_{p3}?)) + pamount(p_4, k_{p4}?))$	
$\delta_{pumps} = \delta_p * (pamount(p_1, works) + (pamount(p_2,$	
$works) + (pamount(p_3, works) + (pamount(p_4, works)))$	

<i>SNormalBroken1</i>	_____
$\Delta_{SteamBoiler1}$	
<i>Input</i>	
<i>ControlInput</i>	
<i>Amount Computation</i>	
$= norm$	
$k_w = broken$	
$k_d = works$	
$s' = s + amount - d?$	
$\delta' = \delta_{pumps} + \delta_d$	
$s' \geq w_{min} + \delta'$	
$s' \leq w_{max} - \delta' \wedge aas' < (w_{min} + w_{max})/2 \rightarrow PumpsControlledOn$	
$s' \geq (w_{min} + w_{max})/2 \rightarrow PumpsControlledOff$	
$v' = closed \wedge a' = on$	
$z' = broken$	

Complete Operation

$ControlNormal1 \hat{=} SnormalFill1$
 $\vee SNormalContinue1$
 $\vee SNormalNotFill1$
 $\vee SNormalWaterStop1$
 $\vee SNormalControlStop1$
 $\vee SNormalBroken1$

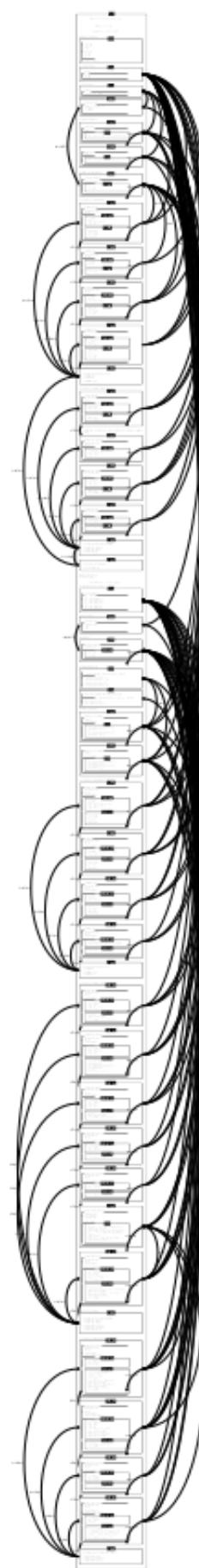
Operations for Broken State

<i>sBrokenContinue1</i>	_____
$\Delta_{SteamBoiler1}$	
<i>Input</i>	
<i>ControlInput</i>	
<i>Amount Computation</i>	
$z = broken$	
$k_w = broken$	
$k_d = works$	
$s' = s + amount - d?$	
$\delta' = \delta + \delta_{pumps} + \delta_d$	
$s' \geq w_{min} + \delta'$	
$s' \leq w_{max} - \delta'$	
$s' < (w_{min} + w_{max})/2 \rightarrow PumpsControlledOn$	
$s' \geq (w_{min} + w_{max})/2 \rightarrow PumpsControlledOff$	
$v' = closed \wedge a' = on$	
$z' = broken$	

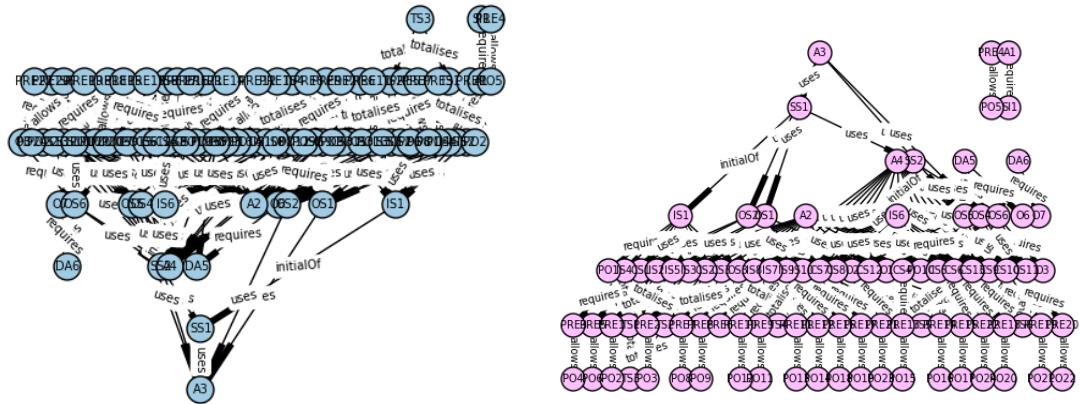
<i>SBrokenNormal1</i>	_____
$\Delta_{SteamBoiler1}$	
<i>Input</i>	
<i>ControlInput</i>	
<i>AmountComputation</i>	
$z = broken$ $k_w = works$ $w? \geq w_{min}$ $w? \leq w_{max}$ $w? < (w_{min} + w_{max})/2 \rightarrow PumpsControlledOn$ $w? \geq (w_{min} + w_{max})/2 \rightarrow PumpsControlledOff$ $s' = w?$ $v' = closed \wedge a' = on$ $z' = norm$	
<i>SBrokenControlStop1</i>	_____
$\Delta_{SteamBoiler1}$	
<i>Input</i>	
<i>ControlInput</i>	
$= broken$ $k_w = broken$ $k_d = broken$ $a' = off \wedge z' = stop$	
<i>SBrokenWaterStop</i>	_____
$\Delta_{SteamBoiler1}$	
<i>Input</i>	
<i>ControlInput</i>	
<i>AmountComputation</i>	
$z = broken \vee z = norm$ $k_w = broken$ $k_d works$ $s' = s + amount - d?$ $z = broken \rightarrow \delta' = \delta + \delta_{pumps} + \delta_d$ $z = norm \rightarrow \delta' = \delta + pumps + \delta_d$ $s' < w_{min} + \delta' \vee s' > w_{max} - \delta'$ $a' = off \wedge z' = stop$	

$ControlBroken1 \hat{=} SBrokenContinue1$
 $\vee SBrokenNormal1$
 $\vee SBrokenControlStop1$
 $\vee SBrokenWaterStop$

13.7.2 ZCGa and ZDRA Output



13.7.3 Dependency and Goto Graphs



13.7.4 General Proof Skeleton

This is written in multi columns.

axiom A1	output 02
stateInvariants SI1	changeSchema CS5
axiom A2	precondition PRE5
axiom A3	postcondition P06
stateSchema SS1	changeSchema CS4
initialSchema IS1	precondition PRE4
postcondition P01	postcondition P05
changeSchema CS7	changeSchema CS6
precondition PRE8	precondition PRE7
postcondition P09	postcondition P08
changeSchema CS2	changeSchema CS1
precondition PRE2	precondition PRE1
postcondition P03	postcondition P02
outputSchema OS1	changeSchema CS3
output 01	precondition PRE3
outputSchema OS3	postcondition P04
precondition PRE6	totaliseSchema TS2
outputSchema OS2	totaliseSchema TS1

totaliseSchema TS3	output 05
axiom A4	changeSchema CS20
stateSchema SS2	precondition PRE21
initialSchema IS2	postcondition P023
postcondition P010	changeSchema CS21
axiom A5	precondition PRE22
axiom A6	postcondition P024
outputSchema OS5	changeSchema CS19
output 04	precondition PRE20
outputSchema OS4	postcondition P022
output 03	changeSchema CS18
changeSchema CS9	precondition PRE19
precondition PRE10	postcondition P021
postcondition P012	changeSchema CS10
changeSchema CS8	precondition PRE11
precondition PRE9	postcondition P013
postcondition P011	changeSchema CS12
changeSchema CS11	precondition PRE13
precondition PRE12	postcondition P015
postcondition P014	changeSchema CS17
changeSchema CS13	precondition PRE18
precondition PRE14	postcondition P020
postcondition P016	changeSchema CS16
changeSchema CS15	precondition PRE17
precondition PRE16	postcondition P019
postcondition P018	totaliseSchema TS6
changeSchema CS14	totaliseSchema TS4
precondition PRE15	totaliseSchema TS5
postcondition P017	lemma L_(CS7)
outputSchema OS6	lemma L_(CS2)

lemma L_(CS5)	lemma L_(CS14)
lemma L_(CS4)	lemma L_(CS20)
lemma L_(CS6)	lemma L_(CS21)
lemma L_(CS1)	lemma L_(CS19)
lemma L_(CS3)	lemma L_(CS18)
lemma L_(CS9)	lemma L_(CS10)
lemma L_(CS8)	lemma L_(CS12)
lemma L_(CS11)	lemma L_(CS17)
lemma L_(CS13)	lemma L_(CS16)
lemma L_(CS15)	

Chapter 14

ZMathLang L^AT_EX package

This chapter shows the ZMathLang L^AT_EX package which was implemented to accommodate the labels to annotate the 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[most]{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}{A0A0AO}
\definecolor{Black}{HTML}{000000}
\definecolor{White}{HTML}{FFFFFF}
\definecolor{expression}{HTML}{0FD016}
\definecolor{set}{HTML}{FF5050}
\definecolor{term}{HTML}{3A9FFF}
\definecolor{declaration}{HTML}{808080}
\definecolor{specification}{HTML}{FFE6CF}
\definecolor{text}{HTML}{C55C5}
\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) {
\VARGE
\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 ZGCa weak types
\newcommand{\expression}[1]{%
\fcolorbox{Black}{expression}{\begin{varwidth}
\dimexpr\linewidth-2\fboxsep\$#1\$ \end{varwidth}}
}

\newcommand{\term}[1]{%
\fcolorbox{Black}{term}{\begin{varwidth}
\dimexpr\linewidth-2\fboxsep\$#1\$ \end{varwidth}}
}

\newcommand{\declaration}[1]{%
\fcolorbox{Black}{declaration}{\begin{varwidth}
\dimexpr\linewidth-2\fboxsep\$#1\$ \end{varwidth}}
}

\newcommand{\text}[1]{%
\fcolorbox{Black}{text}{\begin{varwidth}
\dimexpr\linewidth-2\fboxsep\$#1\$ \end{varwidth}}
}

\newcommand{\specification}[1]{%
\fcolorbox{Black}{specification}{\begin{varwidth}
\dimexpr\linewidth-2\fboxsep\$#1\$ \end{varwidth}}
}

\newcommand{\definition}[1]{%
\fcolorbox{Black}{definition}{\begin{varwidth}
\dimexpr\linewidth-2\fboxsep\$#1\$ \end{varwidth}}
}

\endinput
```