FROM FORMAL SPECIFICATION TO FULL PROOF: A STEPWISE METHOD

by

Lavinia Burski

Submitted for the degree of
Doctor of Philosophy

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

March 2016

Acronyms

ASM Abstract state machine.

CGa Core Grammatical aspect.

DRa Document Rhetorical aspect.

GPSa General Proof Skeleton aspect.

Gpsa General Proof Skeleton aspect.

GpsaOL General Proof Skeleton ordered list.

Hol-Z Hol-Z.

IEC International Electrotechnical Commission.

MathLang MathLang framework for mathematics.

PPZed Proof Power Z.

SIL Safety Integrity Levels.

SMT Satisfiability Modulo Theories.

TSa Text and Symbol aspect.

UML Unified Modeling Language.

UTP Unifying theories of programming.

ZCGa Z Core Grammatical aspect.

ZDRa Z Document Rhetorical aspect.

ZMathLang MathLang framework for Z specifications.

Glossary

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

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

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

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

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

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

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

Chapter 1

Overview of ZMathLang

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

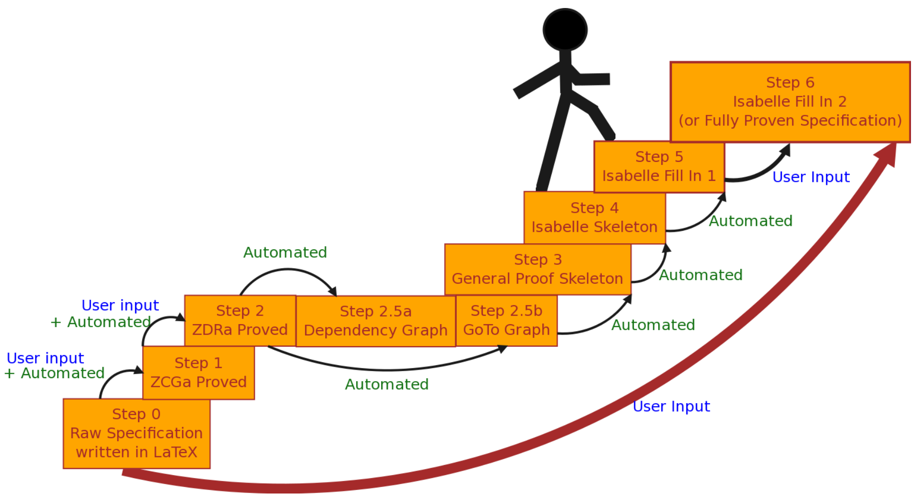


Figure 1.1: The steps required to obtain a full proof from a raw specification.

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

The user doesn't need to go all the way to the top to check for correctness, one advantage of breaking up the translation is that the user gets some level of rigour and can be satisfied with some level of correctness along the way. However the main advantage of breaking up the translation is that the level of expertise needed to check for the correctness of a system specification can be done by someone who has little or no expertise in checking for correctness by a theorem prover.This tool could also aid user in learning theorem proving as it translates their specification and thus they have examples of the syntx used in their theorem prover for their specification. The small black arrows represent the amount of expertise needed for each step. The last step the arrow is slightly thicker as some theorem prover knowledge is needed. However these arrows are still small in comparison to the red thick arrow which represents the translation in one big step.

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

- Step 0: Raw LaTeX Z Specification. Start
- Step 1: Check for Core Grammatical correctness (ZCGa). User Input + Automated
- Step 2: Check for Document Rhetorical correctness (ZDRa). User Input + Automated
- Step 3: Generate a General Proof Skeleton (GPSa). Automated
- Step 4: Generate an Isabelle Skeleton. Automated
- Step 5: Fill in the Isabelle Skeleton. Automated

- Step 6: Prove existing lemmas and add more safety properties if needed. User Input

1.1 How far does the automation go?

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

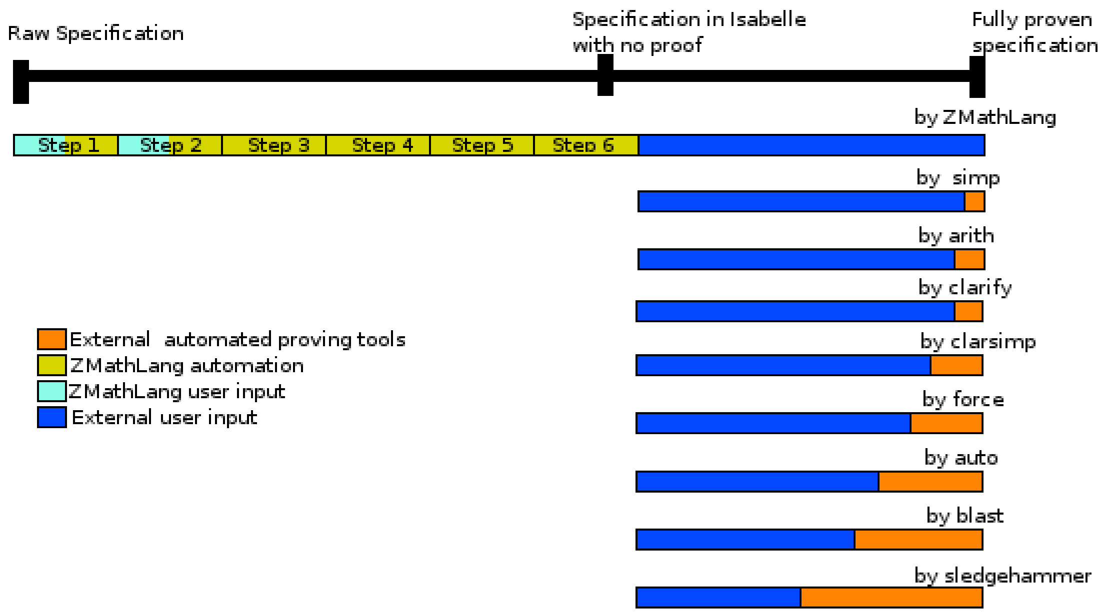


Figure 1.2: How far can one automate a specification proof.

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

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

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

Even external automated theorem provers have their limitations. For example, the user can use the Isabelle tool '*sledgehammer*' assists the user automatically solving some proofs, but not all. The sledgehammer documentation advises to call '*auto*' or '*safe*' followed by '*simp_all*' before invoking sledgehammer. Depending on the complexity of your proof, this sometimes may prove the users properties, other times it may not and the user will still need to invoke sledgehammer to assist in reaching their goal. Sledgehammer itself is a tool that applies Satisfiability Modulo Theories (SMT) solvers on the current goal e.g. Vampire[7], SPASS [6] and E [8]. We use sledghammer as a collective, to describe all the SMT solvers it covers [2].

Other automated methods include:

- **simp:** simplifies the current goal using term rewriting.
- **arith:** automatically solves linear arithmetic problems.
- **clarify:** like auto but less aggressive.
- **clarsimp:** a combination of clarify and simp.
- **force:** like auto but only applied to the first goal.

- **auto:** applies automated tools to look for a solution.

- **blast:** a powerful first-order prover. [5]

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

1.2 Overview of ZMathLang step by step

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

1.2.1 Step 0- The raw LaTeX file

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

$[NAME, DATE]$

BirthdayBook
$known : \mathbb{P}\, NAME$
$birthday : NAME \pfun DATE$

$known = \mathrm{dom}\ birthday$

InitBirthdayBook
$BirthdayBook'$

$known' = \{\}$

AddBirthday
$\Delta BirthdayBook$
$name? : NAME$
$date? : DATE$

Figure 1.3: Example of a partial Z specification.

1.2.2 Step 1- The Core Grammatical aspect for Z

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

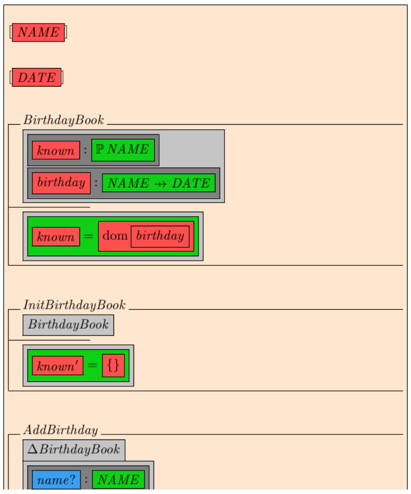


Figure 1.4: Example of a ZCGa annotated specification.

1.2.3 Step 2- The document Rhetorical aspect for Z

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

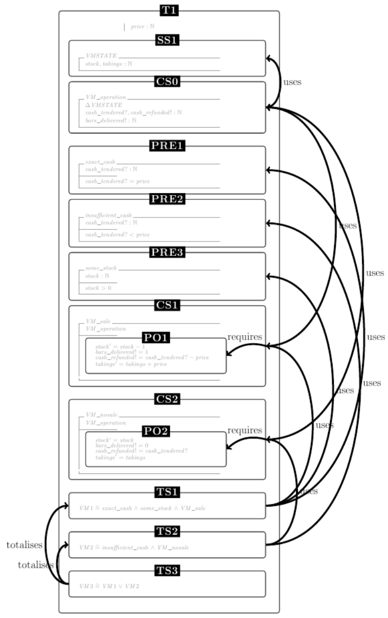


Figure 1.5: Example of a ZDRa annotated specification.

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

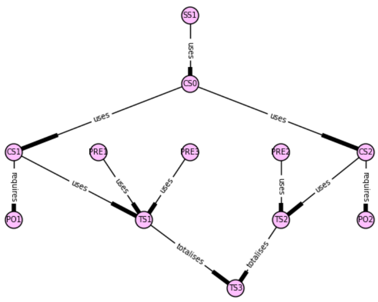


Figure 1.6: Example of an automatically generated goto graph.

1.2.4 Step 3- The General Proof skeleton

The following step is an automatically generated General Proof Skeleton aspect (Gpsa). This document is automated using the goto graph which is generated from the ZDRa annotated LaTeX specification. It uses the goto graph to describe in which logical order to input the specification into any theorem prover. At this stage it also adds simple proof obligations to check for the consitancy of the specification i.e. the specification is not conflictive each part. An example of a general proof skeleton is shown in figure 1.7. The Gpsa is further described in section **??**.

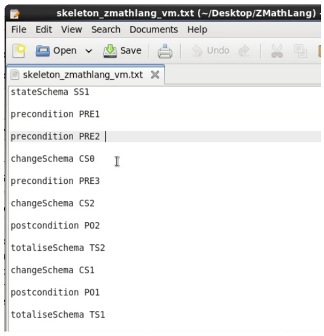
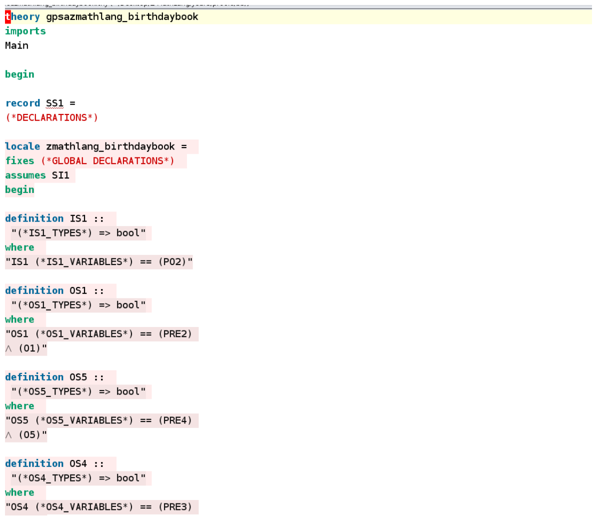


Figure 1.7: Example of a general proof skeleton.

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

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

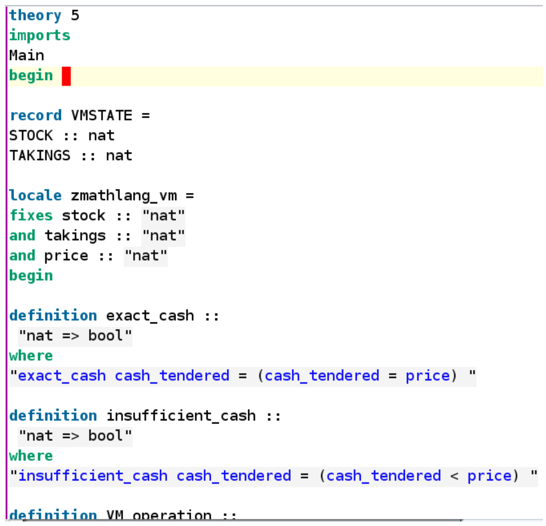


```
theory gpsazmathlang_birthdaybook
imports
Main

begin

record SS1 =
(*DECLARATIONS*)

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

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

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

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

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

Figure 1.8: Example of an Isabelle skeleton.

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

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



```
theory 5
imports
Main
begin

record VMSTATE =
STOCK :: nat
TAKINGS :: nat

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

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

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

definition VM_operation ::
```

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

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

1.2.7 Step 6- A fully proven Z specification

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

1.3 Procedures and products within ZMathLang

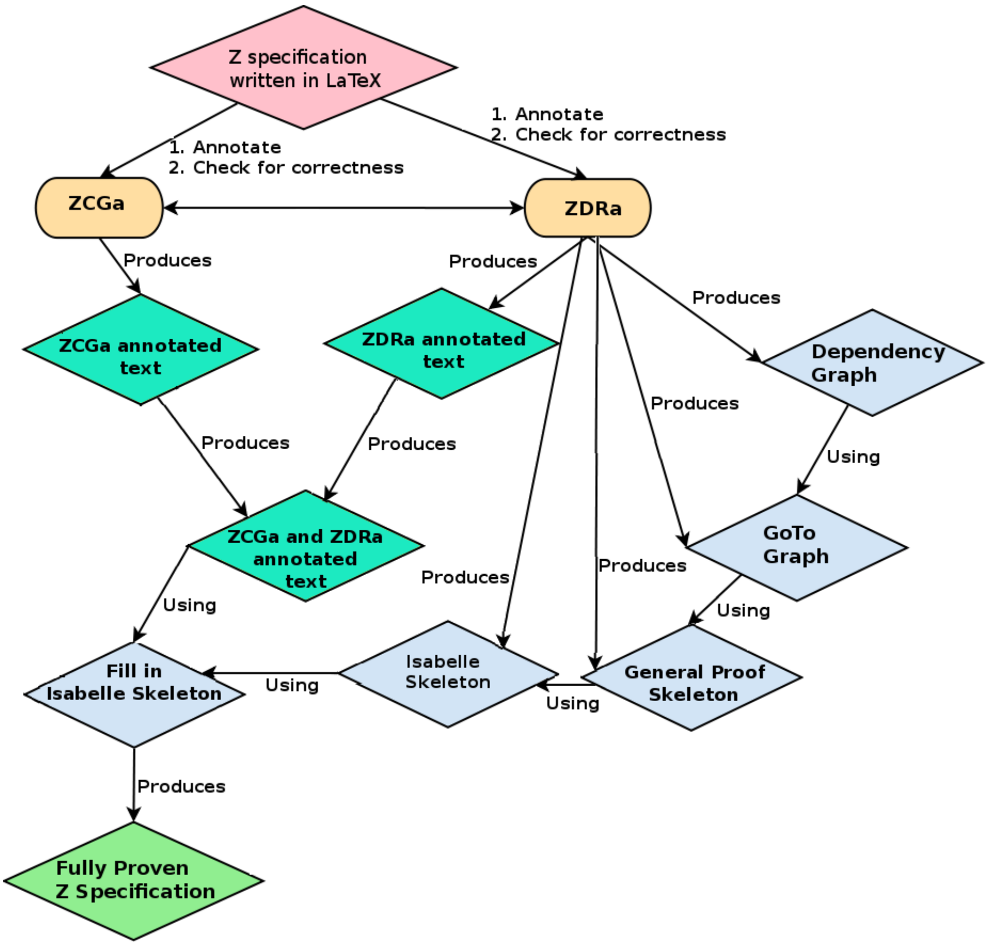


Figure 1.10: Flow chart of ZMathLang.

Figure 1.10 shows a flow chart describing the documents produced from using the framework and which parts are fully automated, partially automated and user input. Products which are created by full automation are diamonds in blue. Diamonds in green are produced by user input and products shown in aqua diamonds are partial automated.

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

1.4 The ZMathLang LaTeX Package

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

1.4.1 Overview

The ZMathLang style file invokes the following packages:

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

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

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

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

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

1.4.2 LaTeX commands to identify ZDRa Instances

The ZDRa section of the LaTeX file provides three new commands: `\draschema`, `\draline` and `\dratheory`. The `\dratheory` annotation is for the entire specification which contins all the intances and relations. The `\draschema` command is to annotate the instances which are entire zed schemas, this command should go before any `\begin{schema}` or `\begin{zed}` command. The `\draline` annotation is to annotate any instance that is a line of text which contains plain text or ZCGa annotated text. But does not include any ZDRa annotated text. For example in figure 1.12 the `\draline{PRE1}` annotation is embedded in the `\draline{CS1}{` which will not compile. Therefore the correct way this schema is labelled is shown in figure 1.13 where the `\draline{PRE1}` annotation is embedded in the `\draschema{CS1}` annotation.

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

Figure 1.12: Incorrect annotating of ZDRa.

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

Figure 1.13: Correct annotating of ZDRa.

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

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

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

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

1.4.3 LaTeX commands to identify ZDRa Relations

There are 5 new commands to define the relations for the ZDRa, these are *initialOf*, *uses*, *totalises*, *requires* and *requires*. Information on these relations are described in chapter **??**, however this section of the thesis describes the LaTeX commands implemented so that they can be used to annotated a specification in ZDRa.

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

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

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

1.4.4 LaTeX commands to identify ZCGa grammatical types

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

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

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

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

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

1.5 Conclusion

In total there are 6 steps in order to translate a Z specification into the theorem prover Isabelle. Each of these steps assist the user in understand the specification more, and some steps even produce documents, graphs and charts in order to analyse the specification. These products also allow others in the development team to understand the system better such as clients, stakeholders, developers etc. The majority of the steps are fully automated whilst some a little user input. The next chapter begins to describe step 1 (ZCGa) in more detail.

Chapter 2

Evaluation and Discussion

In this chapter we go through a few case studies and discuss the difference between the specification translations if any. Table 2.1 shows the specifications we have translated into Isabelle using ZMathLang. We have classified these examples to show the different types of specifications which can be translated using the ZMathLang toolkit. In this chapter we take one example from each class and describe in more detail how the translation was done.

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

Table 2.1: A table showing the specifications we have translated into Isabelle using ZMathLang

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

2.1 Complexity of specifications

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

2.1.1 Raw Latex Count

Table 2 how long each specification is by amount of lines of code and environments uses. We have listed the specifications in decreasing complexity of how many lines of LaTeX the raw specification has.

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

Table 2.2: How many zed, schema and axdef environments and lines of LaTeX code makes up each specification

We list information about how many different envronments and lines of LaTeX make up each specification in table 2.2. The environment numbers count how many different types of environments exist within the specification. That is how many '`\begin{schema}...\end{schema}`' or '`\begin{zed}`... etc. We add up the total amount of environments in the specification. From the table we can see that for most of the specifications the more lines of LaTeX there is then the total amount of environments increse. However, there are three exceptions to this trend. The '*BirthdayBook*' specification, '*ClubState2*' specification and '*Vendine Machine*' specification. Specifications for systems are can always be written in a variety of ways and still have the same meaning. Even formal specifications can be written different ways. For example one may have the following declarations:

$t : \mathbb{N}$

$l : \mathbb{N}$

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

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

Thus removing a line. Formal specifications can also include comments written in natural language which are not part of the formal script. These extra comments about the specification may have also added to the line count in table 2.2.

2.1.2 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 2.3: 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 2.3. We remind the reader the colours corresponding to each grammatical type are: *schematext* , *declaration* , *expression* , *term* , *set* and *definition* . In this instance we don't use specification as we assume each document contains a single specification.

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

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

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

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

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

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

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

The ZCGa text contains 1 annotation of `SchemaText`, 1 annotation of a `declaration`, 2 annotations of `sets` and 1 annotation of an `expression`. Since this is the case we expect to see more expressions than declarations in every specification, which is true according to table 2.3.

2.1.3 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 2.4 and give details of the amount of relations in each specification in table 2.5.

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

Table 2.4: How many of each ZDRa instances exists in each specification.

From table 2.4 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. However this is not a must and some specification (even from our sample) do not have any stateInvariants.

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

Lemma 2.1.1. $precondition \longrightarrow postcondition \vee output$

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

Specification	ZDRa Relations				
	initiaOf	**requires**	**allows**	**totalises**	**uses**
Steamboiler	2	28	21	24	92
ProjectAlloc	1	16	11	0	16
VideoShop	0	15	13	0	142
TelephoneDirectory	1	11	8	14	8
ClubState	1	12	9	14	12
ZCGa	1	9	6	0	7
GenDB	1	8	6	0	6
Timetable	1	6	4	0	6
BirthdayBook	1	7	4	6	5
AutoPilot	0	2	1	0	2
ClubState2	1	6	3	0	6
Vending Machine	0	2	0	2	8
ModuleReg	0	3	2	0	2

Table 2.5: How many of each ZDRa relations exists in each specification.

We can cross reference the table showing the amount of instances (table 2.4) with the table showing the relations (table 2.5). For example, the relation *initialOf* can only occur if the specification has an *initialSchema*. Not all specifictions have an *initialSchema* and therefore do not have an *initialOf* relation.

There is also an equal amount of *allows* relations as there is *preconditions*. As was written previously, all preconditions must have a corresponding output or post-condition, therefore the relation '*allows*' links each precondition to its corresponding postcondition or output. However, the vendingMachine specification is an exception to this as the preconditions are written as entire schema's. For example we have the following instance in the vending machine specification:

```
\draschema{PRE3}{
\begin{schema}{some\_stock}
stock: \nat
```

```
\where
stock > 0
\end{schema}}
```

This chunk of specification describes an entire schema as a precondition. The totalising schema then joints the precondition to their corresponding output or postcondition. The specification is written in this way as it is a personal choice of writing the specification formally. All other specifications in our sample set are written in the style where the precondition and corresponding output or postcondition are written inside the same schema environment.

Obviously, the relation '*totalises*' only occurs in specifications where totaliseSchema's are present. Therefore the '*totalise*' relation is not necessary in all specifications.

VideoShop specification is one of the largest specifications (in terms of lines of LaTeX) in our sample set however it has quite a small amount of relations.

2.2 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.

2.2.1 Case Study 1: A specification using only terms.

The following case study is based on the *Steamboiler* [1] 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 larges from our examples. It is made up of 507 lines of LaTeX code, 10 zed environemnts, 34 schmes 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 post-conditions, 12 outputs and 1 set of stateInvariants.

2.2.1.1 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 controlers. There is a valve for emptying the boiler.

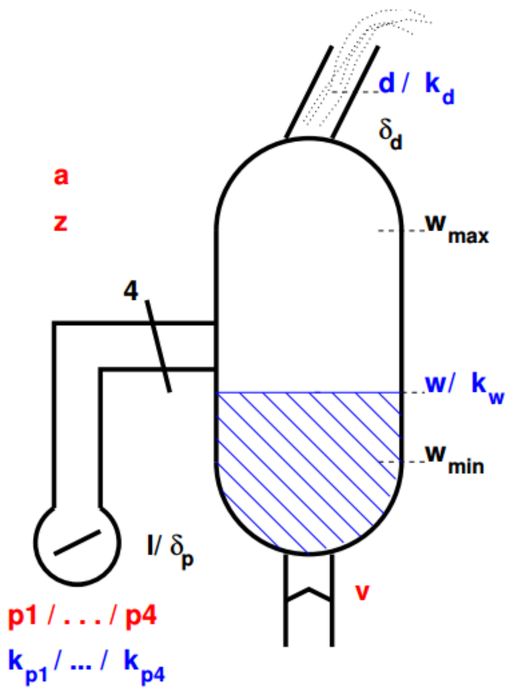


Figure 2.1: A diagram showing a theoretical Steamboiler.

An example of how the steamboiler could look is shown in figure 2.1. The variables of the steamboiler are shown in table 2.6.

find out what l does

variables	**description**
w_{min}	minimal water level
w_{max}	maximal water level
l	
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 2.6: The variables of the steamboiler and their descriptions.

The full formal specification for the steamboiler is 10 pages long which can be found in [3]. Therefore we have given small examples taken from the full specification.

2.2.1.2 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 \\
\where
w_{min} < w_{max}
\end{axdef}

Measured values

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

Figure 2.2: The formal specification LaTeX code for the steamboiler system.

$State ::= init \mid norm \mid broken \mid stop$

$OnOff ::= on \mid off$

$OpenClosed ::= open \mid closed$

hysical Constants

$w_{min} : \mathbb{N}$
$w_{max} : \mathbb{N}$
$l : \mathbb{N}$
$d_{max} : \mathbb{N}$
$\delta_p : \mathbb{N}$
$\delta_d : \mathbb{N}$

$w_{min} < w_{max}$

leasured values

Input
$w? : \mathbb{N}$
$d? : \mathbb{N}$

ontrol values

Pumps
$p_1, p_2, p_3, p_4 : OnOff$

SteamBoiler0
Pumps
$v : OpenClosed$
$a : OnOff$
$z : State$

uxiliary Schemata

PumpsOff
Pumps'

$p_1' = off \land p_2' = off \land p_3' = off \land p_4' = off$

PumpsOn
Pumps'

$p_1' = on \land p_2' = on \land p_3' = on \land p_4' = on$

Figure 2.3: The formal specification for the steamboiler system.

We show the LaTeX code for part of the raw steamboiler specification in figure 2.2 and it's pdflatex counterpart in figure 2.3.

We then annotate the specification using ZCGa and ZDRa labels.

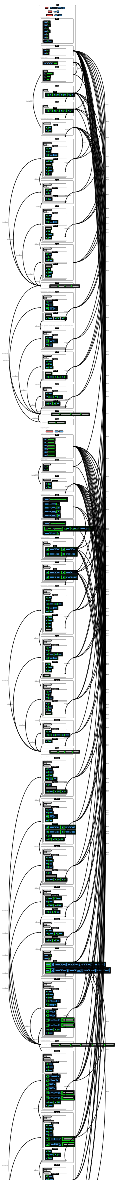


Figure 2.4: An example of the original steamboiler specification annotated in ZCGa and ZDRa.

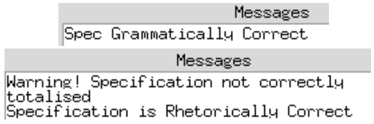


Figure 2.5: The outputting result when checking the steamboiler specification with the ZCGa and ZDRa checkers.

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.

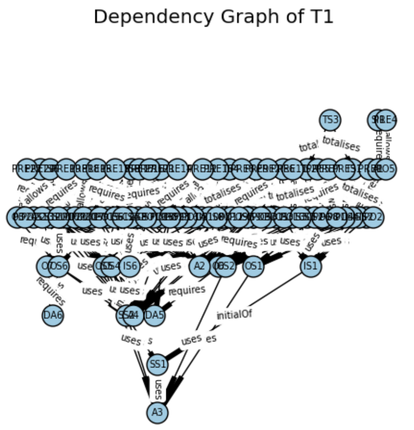


Figure 2.6: The dependecy graph produced for the steamboiler specification.

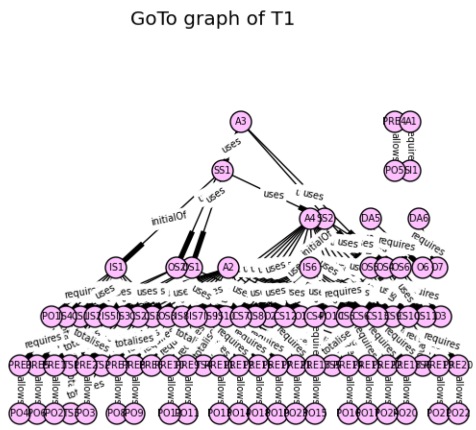


Figure 2.7: The goto graph produced for the steamboiler specification.

The dependecy and goto graphs are shown in figures **??** and 2.7 respectively. Since there are a lot of ZDRa instances and therefore a lot of nodes, both the dependecy graph and goto graph are cluttered. 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 2.8.

```
axiom A1
stateInvariants SI1
axiom A2
axiom A3
stateSchema SS1
initialSchema IS1
postcondition PO1
changeSchema CS7
precondition PRE8
postcondition PO9
changeSchema CS2
```

Figure 2.8: Gpsa for the steamboiler specification.

We can now translate the Gpsa into Isabelle syntax using the ZMathLang toolkit.

```
theory steamboilerSkelton
imports
Main

begin
(*DATATYPES*)

record SS1 =
(*DECLARATIONS*)

locale 1n2 =
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)"
```

```
lemma CS7_L1:
"(∃ (*CS7_VARIABLESANDTYPES*).
(PRE8)
∧ (P09)
⟶ ((SI1)
∧ (SI1')))"
sorry

lemma CS2_L2:
"(∃ (*CS2_VARIABLESANDTYPES*).
(PRE8)
∧ (P09)
⟶ ((SI1)
∧ (SI1')))"
sorry

lemma CS5_L3:
"(∃ (*CS5_VARIABLESANDTYPES*).
(PRE5)
∧ (P06)
⟶ ((SI1)
∧ (SI1')))"
```

Figure 2.9: Part of the isabelle skeleton for the steamboiler specification.

Part of the isabelle skeleton for the steamboiler specification is shown in figure 2.9. Since the steamboiler example has 2 stateSchema's the ZMathLang toolset 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 decribed 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.
```

```
∨ (ControlNormal0 steamboiler0 a' steamboiler0' z' v' p_1' p_2' p_3' p_

end

record SteamBoiler1 = SteamBoiler0 +

s :: "nat"
delta :: "nat"


definition (in  thesteamboiler) SteamBoilerInit1 ::
 "SteamBoiler0 ⇒ nat ⇒ nat ⇒ SteamBoiler0 => OnOff ⇒ State ⇒ bool"
where
```

Figure 2.10: Part of the filled in isabelle skeleton for the steamboiler specification.

In figure 2.10 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. Middway through the theory file we see another record, `SteamBoiler1` which was $SS2$ in ZDRa. This is shown in the bottom picture in figure 2.10. $SS2$ 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. 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 2.11: Manually proven lemma for the steamboiler specification.

Using the lemma's which have been generated in figure 2.10 we have proved all of these lemmas for the steamboiler specification, part of which is shown in figure 2.11. 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 2.11) 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 [3].

2.2.2 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 LaTeX 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 **??**.

2.2.3 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 [4] 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 2.12: An example of the original Autopilot specification.

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

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

$$events ::= press\_att\_cws \mid press\_cas\_eng \mid press\_alt\_eng \mid press\_fpa\_sel$$

Only one of the first three modes can be engaged at any time. However, the cas_eng mode can be engaged at the same time as any of the other modes. The pilot engages a mode by pressing the corresponding button on the panel. One of the three modes, att_cws, fpa_sel, or alz_eng, should be engaged at all times. Engaging any of the first three modes will automatically cause the other two to be disengaged since only one of these three modes can be engaged at a time.

$$mode\_status ::= off \mid engaged$$

off_eng

$mode : mode\_status$

$mode = off \lor mode = engaged$

AutoPilot

Figure 2.13: An example of the Autopilot specification partially formalised.

2.2.3.1 ZMathLang steps for the autopilot case study.

We give the informal specification in figure 2.12 and one which we are beginning to formalised in figure 2.13. We have highlighted in red the parts which we have formalised in figure 2.13. 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.

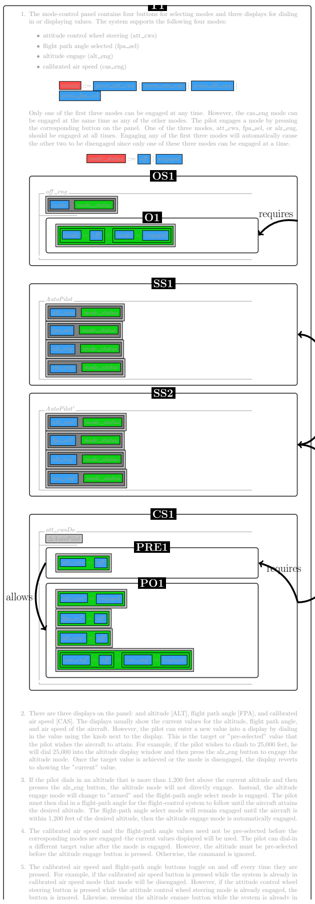


Figure 2.14: An example of the original Autopilot specification annotated in ZCGa and ZDRa.

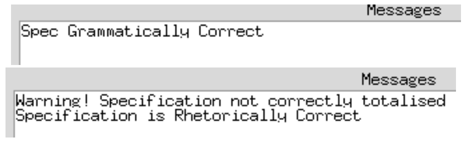


Figure 2.15: The outputting result when checking the autopilot specification with the ZCGa and ZDRa checkers.

Even though the specification is not fully formalised we can still annotate it with ZCGa and ZDRa and check for the correctness of the parts which have been annotated (shown in figures 2.14 and 2.15). When checking with ZDRa we have a warning message telling the user that the specification is not correctlty totalised. That is there is a precondition outstand with not postcondition counter part. This does not matter for now as we can still carry on with the tranlation.

When checking the specification for ZDRa, ZMathLang has also produced a dependecy graph and goto graphs (shown in figures 2.16 and 2.17):

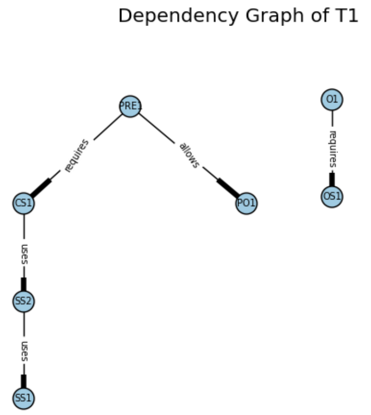


Figure 2.16: The dependecy graph produced for the autopilot specification.

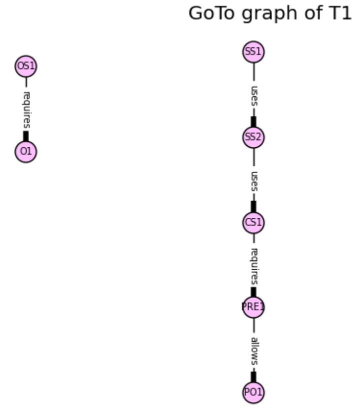


Figure 2.17: The goto graph produced for the autopilot specification.

With the dependency graph (figure 2.16) we can say that $SS2$ uses $SS1$, $CS1$ uses $SS2$, $PRE1$ requires $CS1$ and allows $PO1$. Which makes up the main tree dependencies. $OS1$ and $O1$ are seperate as they do not have any relations which any parts of the main tree, the only dependency they have is on eachother where $O1$ requires $OS1$.

We can say that the dependecy graph *describes* the relation between instances and the goto graph (figure 2.17) *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 2.18). We can clearly see that the arrow has changed direction for the $OS1$ and $O1$ relationship from the dependency graph. Again since these two instances are not dependency on any part of the main tree they are seperate. However in the dependency graph described the relation that $O1$ *requires* $OS1$ ($O1$ root and $OS1$ child) the goto graph flips this relationship as in a theorem prover we woud need $OS1$ to appear before $O1$ since $O1$ requires $OS1$ to exist. We can also say that $SS2$ uses $SS1$ therefore $SS2$ needs $SS1$ to exist for itself to exist. Then $CS1$ uses $SS2$ therefore $CS1$ needs $SS2$ to exist for itself to exit. We can say that $PRE1$ requires $CS1$ and allows $PO1$. Thefore $PO1$ needs $PRE1$ to exist before it is allowed to exist itself.

```
stateSchema SS1
outputSchema OS1
output O1
stateSchema SS2
changeSchema CS1
precondition PRE1
postcondition PO1
```

Figure 2.18: 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 consistancy accross the specification and thus need state invariants to be present.

```
theory gpsa1n2
imports
Main

begin
(*DATATYPES*)

record SS1 =
(*DECLARATIONS*)

locale 1n2 =
fixes (*GLOBAL DECLARATIONS*)
begin

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

definition CS1 ::
"(*CS1_TYPES*) => bool"
where
"CS1 (*CS1_VARIABLES*) ==
 (PRE1)
∧ (PO1)"

end
end
```

Figure 2.19: 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' = off ∨
 cas_eng' = engaged)"

end
end
```

Figure 2.20: The autopilot specification in Isabelle syntax.

ZMathLang can automatically translate the Gpsa into Isabelle syntax (figure 2.19), 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 2.20) we have the annotated specification in Isabelle form. This can now give the user an idea of how to input their specifiation 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 consistancy have been generated. The user can add the state Invariants in their raw LaTeX specification, or fully formalise their specification. Another way to fully prove their specification is to add other properties to the Isabelle document.

2.3 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 1.2 in chapter 1. ZMathLang is able assit 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 2.2.1) 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.

2.3.1 SteamBoiler

In our largest example there were 21 changeSchema's and 1 set of stateInvariants, therefore there was 21 properties which the user had to prove manually.

To prove the sanity of the steamboiler specification, we went through the consistency lemmas (automatically generated) one by one and manually prove them. We started of with unproven lemma's with the command '*sorry*' at the end such as the lemma shown in figure 2.21.

```
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
```

Figure 2.21: The ‘`SNormalStop0`’ lemma taken from the steamboiler halfbaked proof.

We then delete the ‘*sorry*’ command and if sledgehammer is set up to be automatic, the user can sometimes leave their cursor at the end of the lemma and ‘*auto sledgehammer*’ finds a proof which is displayed in the output terminal. In our case, this has happened for the ‘`SNormalStop0`’ lemma which is displayed in figure 2.22.

```
proof (prove): depth 0

goal (1 subgoal):
 1. ∃steamboiler0 a' steamboiler0' w_max' w_min' z' v'.
       z = norm ∧ (w < w_min ∨ w_max < w) ∧ a' = off ∧ z' = stop ⟶
       w_min < w_max ∧ w_min' < w_max'
Auto Sledgehammer ("cvc4") found a proof: by (smt State.distinct(9)).
```

Figure 2.22: Auto sledgehammer finding a proof for one of the lemma’s in the steamboiler specification using the SMT solver ‘*cvc4*’.

In this particular lemma we are proving the property that the `SNormal_Stop)` schema does not conflict with the state invariants of the specification either before or after the state has been changed. Some other lemma’s in the steamboiler example (such as the `SInitNormal1_L9` lemma) could be proven by the isabelle command ‘`blast`’ as shown in figure 2.23.

```
∃ p_2' :: OnOff.
∃ p_1' :: OnOff.
∃ steamboiler0' :: SteamBoiler0.
∃ w_max' :: nat.
∃ w_min' :: nat.
∃ z' :: State .
∃ v' :: OpenClosed.
∃ a' :: OnOff.
(z = init)
∧ (d = 0)
∧ (k_w = works ∧
 k_d = works)
∧ (w ≥ w_min + d_max)
∧ (w ≤ w_max)
∧ (z' = norm)
∧ (v' = closed)
∧ (a' = on)
∧ (s' = w)
∧ ((PumpsOff p_4' steamboiler0 p_3' p_2' p_1' steamboiler0'))
⟶ (w_min < w_max
∧ (w_min' < w_max')))"
by blast
```

Figure 2.23: An example of a lemma in the steamboiler specification being proved by blast.

Proving by '*blast*' is obviously less complex then the proof needed for lemma SNormalStop0_L1 in figure 2.21, however if we look back to figure 1.2 in chapter 1 we can see that '*blast*' covers less properties then '*sledgehammer*'. Therefore for the lemma's in the steamboiler specification we have proved 2 lemma's by blast and 19 using sledgehammer.

It is important to note that a single lemma can be proven in a variety of ways, so even thoguh we have chosen to prove our specification in certain ways other users may choose to use other tools to prove their theorems and lemmas. Even though we have proved 19 lemmas by sledgehammer in the steamboiler specification, we might have been able to prove all the lemmas by sledgehammer but chosen to prove 2 by blast to show variety.

2.3.2 ModuleReg

The modulereg is one of our smallest examples, however with 1 set of stateinvariants and 2 changeSchemas, ZMathLang automatcally produces 2 lemma's to check the sanity of the specification. An example of one of these lemmas is shown in figure 2.24. 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:
"(∃ 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'))"
sorry
```

Figure 2.24: 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 curser 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 2.25). 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. ∃degModules students taking p degModules' students' taking' m.
       (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'
Auto Sledgehammer ("cvc4") found a proof: by (smt Domain_empty Domain_insert Range.intros 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 2.25: Output shown when proving the lemma ‘`RegForModule` shown in figure 2.24 .

By clicking on the auto solving method shown in figure 2.25 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 2.26: 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. 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.

2.3.3 Vending Machine

The vending machine example has 3 state changing schemas (shown in table 2.4) 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 1.2 in chapter 1 it shows that the ZMathLang toolkit 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 toolkit 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.

2.3.4 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.

We show the amount of lemmas and total amount of tactics needed to prove the sanity of each specification in table 2.7. Some lemma's need only one tactic

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 2.7: A table to show the amount of automatically generated lemmas and total amount of tactics used for each specification.

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 Isabelles '*sledgehammer*' tool to assist us with our proving.

2.3.4.1 SteamBoiler

The steamboiler has in total 20 lemmas to check the specifications sanity as it has 20 changing state schemas. The specification has 1 set of stateInvariants and we used 41 tactics to prove our lemmas.

The steamboiler state invariants is shown in the expression part of the stateSchema. The state invariants for this specification is that the minimum water level (`w_min`) must be smaller than the maximum water level (`w_max`). Therefore, the state invariants are writter after '`assumes`' in the specification (`w_min < w_max`).

When ZMathLang produces the proof obigations to check the sanity of the specification it must also check that the changeSchema does not conflict with the prime state invariants. Thus, the proof obligations to check for all changing state schema will check that the preconditions and postconditions of each changeSchema still imply that (`w_min < w_max`) **and** (`w_min' < w_max'`).

For example to prove our first lemma `SNormalStop0_L1`, we wish to make sure that the SNormalStop0 changeSchema does not conflict with the stateInvariants and the stateInvariants prime.

lemma (in thesteamboiler)	begin
SNormalStop0_L1:	Name of lemma
"(\<exists> steamboiler0 ::	
SteamBoiler0.	list of all
\<exists> a' :: OnOff.	variables and
\<exists> steamboiler0' ::	types
SteamBoiler0.	
\<exists> w_max' :: nat.	
\<exists> w_min' :: nat.	
\<exists> z' :: State.	
\<exists> v' :: OpenClosed.	
(z = norm)	preconditions **and**
\<and> (w < w_min	postconditions of
w > w_max)	SNornalStop0 schema
\<and> (a' = off \<and>	
z' = stop)	
\<longrightarrow> (w_min < w_max	**implies** stateInvariants
\<and> (w_min' < w_max')))"	**and** stateInvariants prime
by (smt State.distinct(9))	

Since there are 2 records in our steamboiler specification we have to manually write the line `(in thesteamboiler)` which lets isabelle know that the changeSchema uses variables from the first record. The `SNormalStop0_L1` lemma is proved using 2 tactics `smt` and `State.distinct(9)`. We remind the reader that although we have used these 2 tactics to prove our lemma there may be other ways in which to do so.

Our most complex proof is for the `SNormalContinue1_L11` lemma in which we use 3 tactics to prove the property that the `SNormalContinue1` changeSchema does not conflict with the state Invariants. The tactics we use here are `smt`, `OnOff.distinct(1)` and `pswitch.simps(1)`.

2.3.4.2 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 consistancy. To prove these lemmas we have in total used 18 tactics.

The lemmas to check that the changeSchema's did not conflict with thestateInvariants with were as follows:

```
\<exists> variables and types.
Preconditions
\<and>
PostConditions
\<longrightarrow>
(((dom studInterests) \<inter> (dom lecInterests) = {})
\<and> (dom allocation \<subseteq> dom studInterests)
\<and> (ran allocation \<subseteq> 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' \<subseteq> dom studInterests')
\<and> (ran allocation' \<subseteq> 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).

2.3.4.3 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 \<subseteq> members)
\<and> (Range rented \<subseteq> dom stockLevel)
\<and> (\<forall> t \<in> Range rented.
 card ({p. (p, t) \<in> rented}) < (the (stockLevel t)))
\<and> (Domain rented' \<subseteq> members')
\<and> (Range rented' \<subseteq> 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`.

2.3.4.4 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 consistancy 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.

2.3.4.5 ClubState

The clubstate specification has 4 changing schemas and thus 4 properties to check the stateinvairants 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 \<subseteq> badminton)
\<and> (card hall \<leq> maxPlayers)
\<and> (hall' \<subseteq> 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`.

2.3.4.6 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 \<subseteq> declarations)
\<and> (SetDeclaration \<subseteq> declarations)
\<and> (dvars \<subset> sets \<union> terms)
\<and> (sets \<inter> terms = {})
\<and> (TermDeclaration' \<subseteq> declarations')
\<and> (SetDeclaration' \<subseteq> 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`.

2.3.4.7 GenDB

GenDB has 4 consistancy 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 \<subseteq> dom sex
\<and> (\<forall>p :: PERSON. (p, p) \<notin> parent^*)
\<and> (\<forall>p :: PERSON. \<forall>q :: PERSON.
\<forall>r :: PERSON. ({(p,q),(p,r)} \<subseteq> parent)
\<and> q \<noteq> r \<longrightarrow>
the (sex q) \<noteq> the (sex r)))
\<and> (Domain parent’ \<union> Range parent’ \<subseteq> dom sex’
\<and> (\<forall>p :: PERSON. (p, p) \<notin> parent’^*)
\<and> (\<forall>p :: PERSON. \<forall>p :: PERSON.
\<forall>r :: PERSON. ({(p,q),(p,r)} \<subseteq> parent’)
\<and> q \<noteq> r \<longrightarrow>
the (sex’ q) \<noteq> the (sex’ r))))"
```

Out of the 4 lemms we proved, 2 lemms can be proven by the tactic `blast`. The other 2 lemms 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.

2.3.4.8 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 timtable 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`.

2.3.4.9 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'))"
```

2.3.4.10 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 \<subseteq> badminton)
\<and> (card hall \<leq> maxPlayers)
\<and> [onCourt, (Range waiting)] partition [hall]
\<and> (hall' \<subseteq> badminton')
\<and> (card hall' \<leq> maxPlayers)))
\<and> [onCourt', (Range waiting')] partition [hall']"
```

The stateInvairants taken from the ClubState2 state schema is

`[onCourt, (Range waiting)] partition [hall]` however the ClubState2 state schema uses the ClubState schema. Therefore the original stateInvairants

(`(hall \<subseteq> 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`).

2.3.4.11 ModuleReg

The ModuleReg specification has 2 changeSchemas and 1 set of stateInvairants. therefore ZMathLang has produced 2 lemmas which we need to prove.

```
\<exists> variables and types.
Preconditions
\<and>
PostConditions
\<longrightarrow>
 (Domain taking \<subseteq> students)
\<and> (Range taking \<subseteq> degModules)
```

```
\<and> (Domain taking' \<subseteq> students')
\<and> (Range taking' \<subseteq> 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'
```

2.4 Reflection and Discussion

To check the consistancy 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 toolkit can take us,how difficult it is for the user to get the full proof and what other proof is left.

2.4.1 How far can ZMathLang toolkit take us and what is left.

Looking back at our examples described in the previous section. The proofs to check for the complexity depend on the complexity of the specification. 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. 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 1.2 in chapter 1, `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 1.2) 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. Thefore a big chunk of work has been done. The advantage with this approach goes to the system designer as they do not need to learn anything beyond their existing knowledge. However, the disadvantage to this approach is that 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 toolkit will assist the designer translating the specification into Isabelle syntax, however the user will then need to learn more Isabelle to finish of 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. The advantage of this scenario is that since 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. The disadvantage in this case is that 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, an Isabelle expert. The system designer translates the specification into Isabelle using the ZMathLang toolkit 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 consistancy using the automated tool. However some lemmas if they are more complex may require more Isabelle skills then 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 remainer of the proof. If the stakeholders require more properties proved then the Isabelle expert can do this as well.

The third case would be most beneficial as an Isabelle expert may not be required if the stakeholders only want to check the system for consistancy and the syntax of the specification is reletively 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 rigourous proof.

2.4.2 Assumptions and limitations of the ZMathLang toolkit

Assumptions

- if 2 records (2 state schema). need to manually write the line to whichever the lemma needs
- Specification = 1 theory. Can't do more than 1 specification in 1 document.

- we assume the user wishes to check for consistancy. As stated in 2proofs, the properties to prove is down to stakeholders

Limitations

- If we totalise preconditions e.g `\totalises{TS#}{PRE#}` and all preconditions have been totalised then no warning. If we totalise schemas with preconditions within them e.g. `\totalises{TS#}{CS#}` then the ZDRa checker doesnt pick up on it.
- when viewing goto and dep graph if there are a lot of nodes they are all bundled together. Would be better if spaced out more.

2.5 Conclusion

Complete Evaluation and discussion chapter

Bibliography

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

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

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

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

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

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

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

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