

**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

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.

Acknowledgements

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

Contents

1	Overview of ZMathLang	1
1.1	How far does the automation go?	3
1.2	Overview of ZMathLang step by step	5
1.2.1	Step 0- The raw LaTeX file	5
1.2.2	Step 1- The Core Grammatical aspect for Z	6
1.2.3	Step 2- The document Rhetorical aspect for Z	7
1.2.4	Step 3- The General Proof skeleton	8
1.2.5	Step 4- The Z specification written as an Isabelle Skeleton	9
1.2.6	Step 5- The Z specification written as in Isabelle Syntax	10
1.2.7	Step 6- A fully proven Z specification	11
1.3	Procedures and products within ZMathLang	12
1.4	The ZMathLang LaTeX Package	13
1.4.1	Overview	13
1.4.2	L ^A T _E X commands to identify ZDRa Instances	14
1.4.3	L ^A T _E X commands to identify ZDRa Relations	16
1.4.4	L ^A T _E X commands to identify ZCGa grammatical types	16
1.5	Conclusion	17
2	Evaluation and Discussion	18
2.1	Complexity of specifications	19
2.1.1	Raw Latex Count	19
2.1.2	ZCGa Count	21
2.1.3	ZDRa Count	23
2.2	Case Studies	26

2.2.1	Case Study 1: A specification using only terms.	26
2.2.1.1	Natural Lanaguge Specification of the Steamboiler	27
2.2.1.2	ZMathLang steps for the steamboiler case study.	29
2.2.2	Case Study 2: A specification using both terms and sets.	35
2.2.3	Case Study 3: A semi formal specification.	35
2.2.3.1	ZMathLang steps for the autopilot case study.	36
2.3	Analysing examples	41
2.3.1	SteamBoiler	41
2.3.2	ModuleReg	44
2.3.3	Vending Machine	46
2.3.4	Other examples	46
2.4	Reflection and Discussion	47
2.4.1	How far can ZMathLang toolkit take us and what is left.	47
2.4.2	Assumptions and limitations of the ZMathLang toolkit	47
2.5	Conclusion	47
Bibliography		48

List of Tables

2.1	A table showing the specifications we have translated into Isabelle using MathLang framework for Z specifications (ZMathLang)	18
2.2	How many zed, schema and axdef environments and lines of \LaTeX code makes up each specification	20
2.3	How many of each grammatical category exists in each specification.	21
2.4	How many of each ZDRa instances exists in each specification. . . .	24
2.5	How many of each ZDRa relations exists in each specification.	25
2.6	The variables of the steamboiler and their descriptions.	28

List of Figures

1.1	The steps required to obtain a full proof from a raw specification. . .	1
1.2	How far can one automate a specification proof.	3
1.3	Example of a partial Z specification.	6
1.4	Example of a ZCGa annotated specification.	7
1.5	Example of a ZDRa annotated specification.	8
1.6	Example of an automatically generated goto graph.	8
1.7	Example of a general proof skeleton.	9
1.8	Example of an Isabelle skeleton.	10
1.9	Example of an Isabelle skeleton automatically filled in.	11
1.10	Flow chart of ZMathLang.	12
1.11	Part of the syntax to define the colours for Z Core Grammatical aspect (ZCGa) in the ZMathLang \LaTeX file.	14
1.12	Incorrect annotating of Z Document Rhetorical aspect (ZDRa). . . .	15
1.13	Correct annotating of ZDRa.	15
1.14	The syntax to define a ZDRa schema instance in the ZMathLang \LaTeX file.	15
1.15	The syntax to define a ZDRa schema relation in the ZMathLang \LaTeX file.	16
1.16	The syntax to define a ZCGa grammatical categories.	17
2.1	A diagram showing a theoretical Steamboiler.	27
2.2	The formal specification \LaTeX code for the steamboiler system. . . .	29
2.3	The formal specification for the steamboiler system.	29

2.4	An example of the original steamboiler specification annotated in ZCGa and ZDRa.	30
2.5	The outputting result when checking the steamboiler specification with the ZCGa and ZDRa checkers.	30
2.6	The dependency graph produced for the steamboiler specification.	31
2.7	The goto graph produced for the steamboiler specification.	31
2.8	General Proof Skeleton aspect (Gpsa) for the steamboiler specification.	31
2.9	Part of the isabelle skeleton for the steamboiler specification.	32
2.10	Part of the filled in isabelle skeleton for the steamboiler specification.	33
2.11	Manually proven lemma for the steamboiler specification.	34
2.12	An example of the original Autopilot specification.	36
2.13	An example of the Autopilot specification partially formalised.	36
2.14	An example of the original Autopilot specification annotated in ZCGa and ZDRa.	37
2.15	The outputting result when checking the autopilot specification with the ZCGa and ZDRa checkers.	37
2.16	The dependency graph produced for the autopilot specification.	38
2.17	The goto graph produced for the autopilot specification.	38
2.18	Gpsa for the Autopilot specification.	39
2.19	The Isabelle skeleton produced for the autopilot specification.	40
2.20	The autopilot specification in Isabelle syntax.	40
2.21	The ‘SNormalStop0’ lemma taken from the steamboiler halfbaked proof.	42
2.22	Auto sledgehammer finding a proof for one of the lemma’s in the steamboiler specification using the Satisfiability Modulo Theories (SMT) solver ‘cvc4’.	42
2.23	An example of a lemma in the steamboiler specification being proved by blast.	43

2.24	An example of one of the lemma's to check for consistency in the modulereg specification.	44
2.25	Output shown when proving the lemma 'RegForModule' shown in fig- ure 2.24	45
2.26	The 'RegForModule' lemma proved using Auto sledgehammer meth- ods.	45

Todo list

■ find out what l does	27
■ gendb and projectalloc, lemmas to prove for consitancy	46
■ Complete Evaluation and discussion chapter	47

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 Mathematically 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 C 's 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.

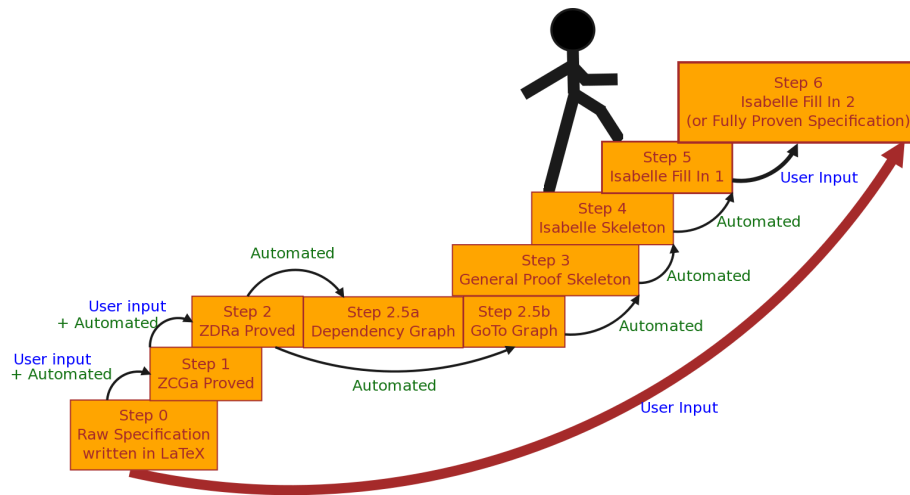


Figure 1.1: The steps required to obtain a full proof from a raw specification.

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 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 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 syntax 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. 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.

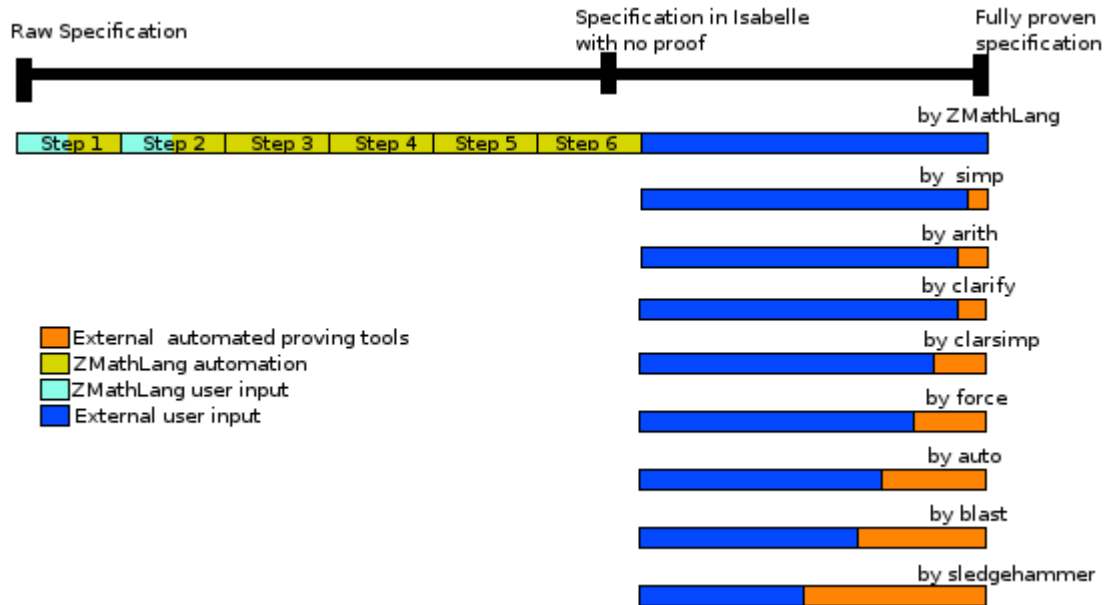


Figure 1.2: How far can one automate a specification proof.

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 (ZCGa and ZDRa) however the rest is automated.

The black line shows the path going from a raw specification to a fully proven specification with a milestone in the middle, which signifies when a specification has been translated into Isabelle syntax but yet has no properties or proof. ZMathLang takes the user a little past this milestone as the toolset also generates properties to check the specification for consistency (see chapter ?? 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 SMT solvers on the current goal e.g. Vampire[67], SPASS [25] and E [70]. We use sledghammer as a collective, to describe all the SMT solvers it covers [9].

Other automated methods include:

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

- **blast**: a powerful first-order prover. [24]

All these automated tools get increasingly further by automating proofs, e.g *clarsimp* covers more proving techniques than *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 individual 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 L^AT_EX 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.

$$[NAME, DATE]$$

$\begin{array}{l} \textit{BirthdayBook} \\ \textit{known} : \mathbb{P} NAME \\ \textit{birthday} : NAME \rightarrow DATE \\ \hline \textit{known} = \text{dom } \textit{birthday} \end{array}$
--

$\begin{array}{l} \textit{InitBirthdayBook} \\ \textit{BirthdayBook}' \\ \hline \textit{known}' = \{\} \end{array}$

$\begin{array}{l} \textit{AddBirthday} \\ \Delta \textit{BirthdayBook} \\ \textit{name}? : NAME \\ \textit{date}? : DATE \end{array}$

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 interchangeable 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 ??.

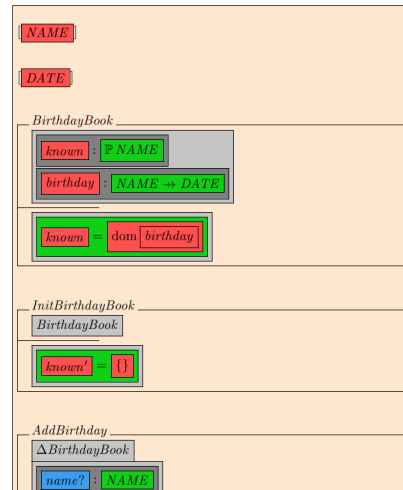


Figure 1.4: Example of a ZCGa annotated specification.

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.

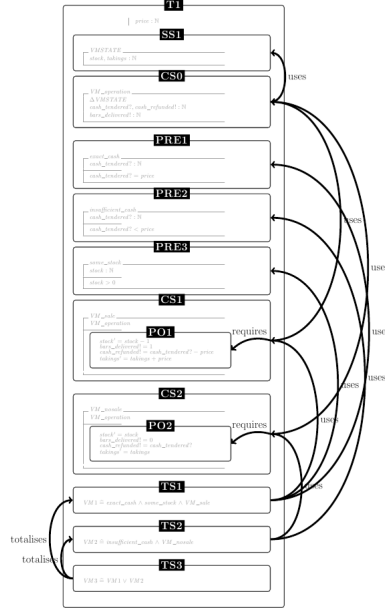


Figure 1.5: Example of a ZDRa annotated specification.

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

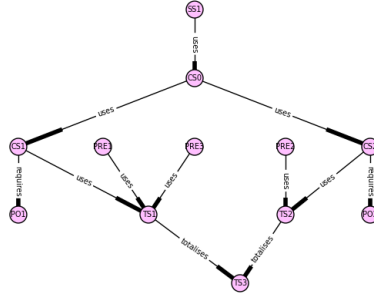


Figure 1.6: Example of an automatically generated goto graph.

1.2.4 Step 3- The General Proof skeleton

The following step is an automatically generated 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 consistency 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 ??.

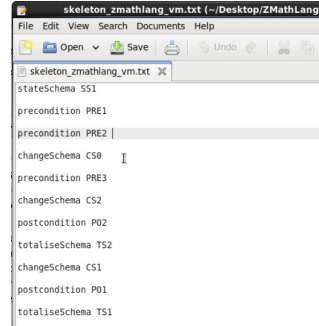


Figure 1.7: Example of a general proof skeleton.

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 gpazmathlang_birthdaybook
imports
Main

begin

record SSI =
(*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)
^ (OS)"

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

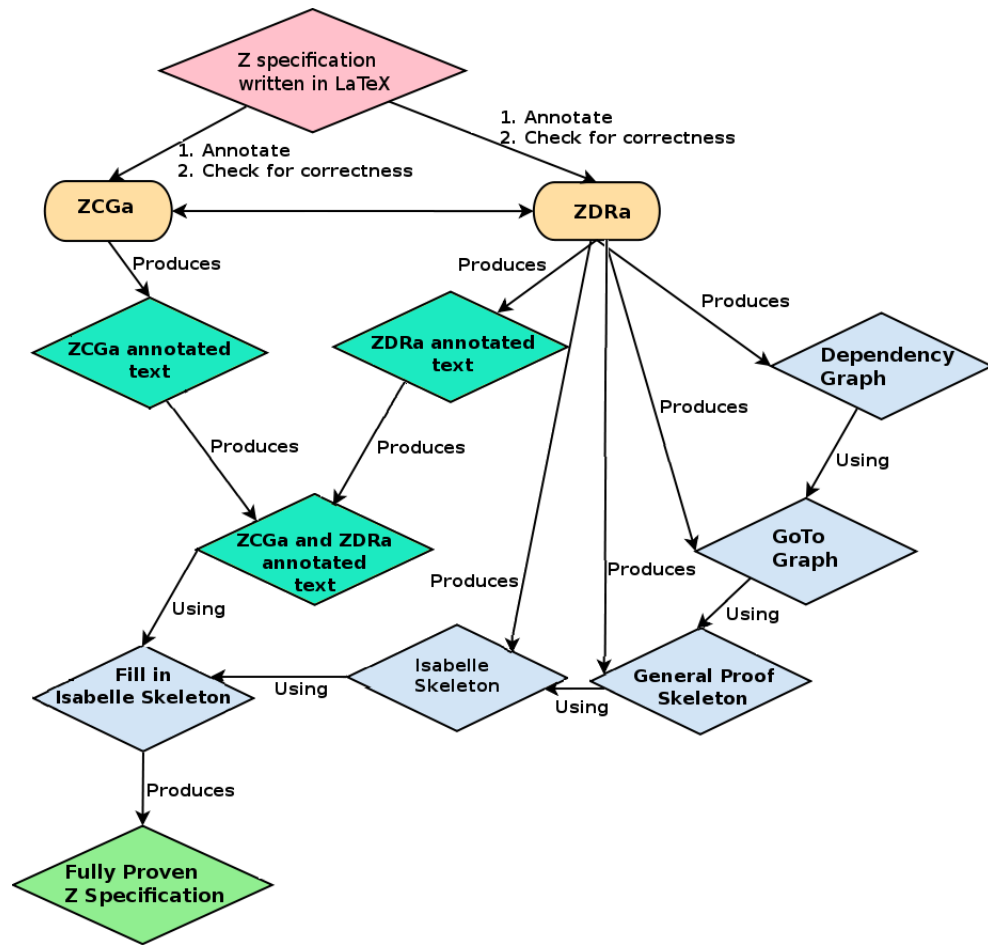


Figure 1.10: Flow chart of ZMathLang.

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 L^AT_EX 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 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. 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.

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

Figure 1.12: Incorrect annotating of ZDRa.

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\fbboxsep\relax}\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 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 the L^AT_EX 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 L^AT_EX 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 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}{\textcolor{green}{\mathit{#1}}}  
}  
  
\renewcommand\set[1]{  
\fcolorbox{Black}{set}{\mathit{#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 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. 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 environments 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 ‘ $\backslash\text{begin}\{\text{schema}\}\dots\backslash\text{end}\{\text{schema}\}$ ’ or ‘ $\backslash\text{begin}\{\text{zed}\}\dots$ ’ 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 increase. However, there are three exceptions to this trend. The ‘*BirthdayBook*’ specification, ‘*ClubState2*’ specification and ‘*Vending Machine*’ specification. Specifications for systems 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 '*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 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 specifications 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 postcondition, 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* [8] 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 envi-

ronemnts, 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.

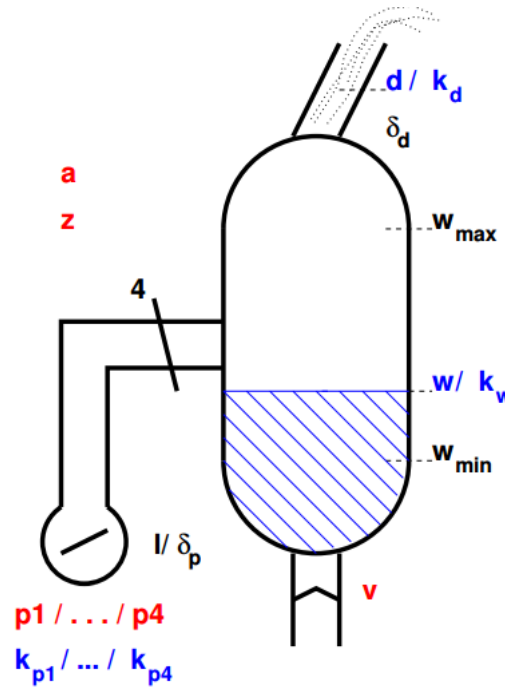


Figure 2.1: A diagram showing a theoretical Steamboiler.

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 [15]. Therefore we have given small examples taken from the full specification.

2.2.1.2 ZMathLang steps for the steamboiler case study.

<pre> \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 \\\ </pre>	<pre> State ::= init norm broken stop OnOff ::= on off OpenClosed ::= open closed hysical Constants w_{min} : \N w_{max} : \N l : \N d_{max} : \N \delta_p : \N \delta_d : \N w_{min} < w_{max} Measured values Input w? : \N d? : \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 \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 </pre>
--	---

Figure 2.2: The formal specification
L^AT_EX code for the steamboiler sys-
tem.

We show the L^AT_EX 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 2.3: The formal specification
for the steamboiler system.

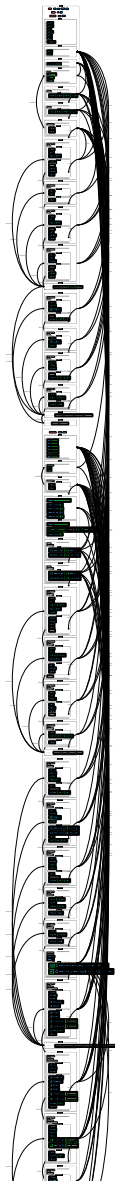


Figure 2.4: An example of the original steamboiler specification annotated in ZCGa and ZDRa.

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.

```

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

```

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

Dependency Graph of T1

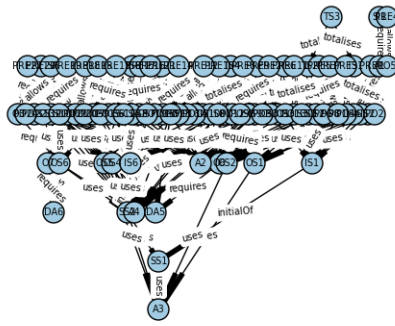


Figure 2.6: The dependency graph produced for the steamboiler specification.

The dependency 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 dependency 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.

GoTo graph of T1

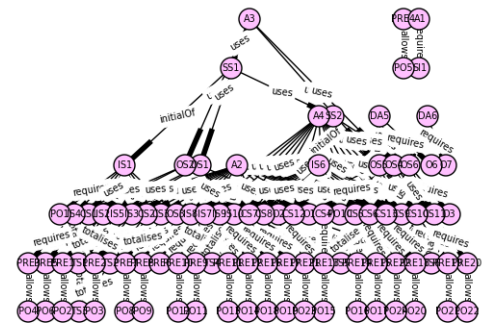


Figure 2.7: The goto graph produced for the steamboiler specification.

```

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 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 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)"

    lemma CS7_L1:
      "( $\exists$  (*CS7_VARIABLESANDTYPES*).
      (PRE8)
       $\wedge$  (P09)
       $\longrightarrow$  ((SI1)
       $\wedge$  (SI1'))))"
    sorry

    lemma CS2_L2:
      "( $\exists$  (*CS2_VARIABLESANDTYPES*).
      (PRE8)
       $\wedge$  (P09)
       $\longrightarrow$  ((SI1)
       $\wedge$  (SI1'))))"
    sorry

    lemma CS5_L3:
      "( $\exists$  (*CS5_VARIABLESANDTYPES*).
      (PRE5)
       $\wedge$  (P06)
       $\longrightarrow$  ((SI1)
       $\wedge$  (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"

end

record SteamBoiler1 = SteamBoiler0 +
s :: "nat"
delta :: "nat"

definition (in thesteamboiler) SteamBoilerInit1 ::
"SteamBoiler0  $\Rightarrow$  nat  $\Rightarrow$  nat  $\Rightarrow$  SteamBoiler0  $\Rightarrow$  OnOff  $\Rightarrow$  State  $\Rightarrow$  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. Midway 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:
  "( $\exists$  steamboiler0 :: SteamBoiler0.
 $\exists$  a' :: OnOff.
 $\exists$  steamboiler0' :: SteamBoiler0.
 $\exists$  w_max' :: nat.
 $\exists$  w_min' :: nat.
 $\exists$  z' :: State .
 $\exists$  v' :: OpenClosed.
  (z = norm)
 $\wedge$  (w < w_min  $\vee$ 
  w > w_max)
 $\wedge$  (a' = off  $\wedge$ 
  z' = stop)
 $\longrightarrow$  (w_min < w_max
 $\wedge$  (w_min' < w_max')))"
by (smt State.distinct(9))

lemma (in thesteamboiler) SInitStop0_L2:
  "( $\exists$  steamboiler0 :: SteamBoiler0.
 $\exists$  a' :: OnOff.
 $\exists$  steamboiler0' :: SteamBoiler0.
 $\exists$  w_max' :: nat.
 $\exists$  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 none 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 [15].

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 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 ??.

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 [16] and started to formalise it.

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

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

Only one of the first three modes can be engaged at any time. However, the cas_eng mode can be engaged at the same time as any of the other modes. The pilot engages a mode by pressing the corresponding button on the panel. One of the three modes, att_cws, fpa_sel, or 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.

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.

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

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 ∨ mode = engaged

AutoPilot

Figure 2.12: An example of the original Autopilot specification.

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.

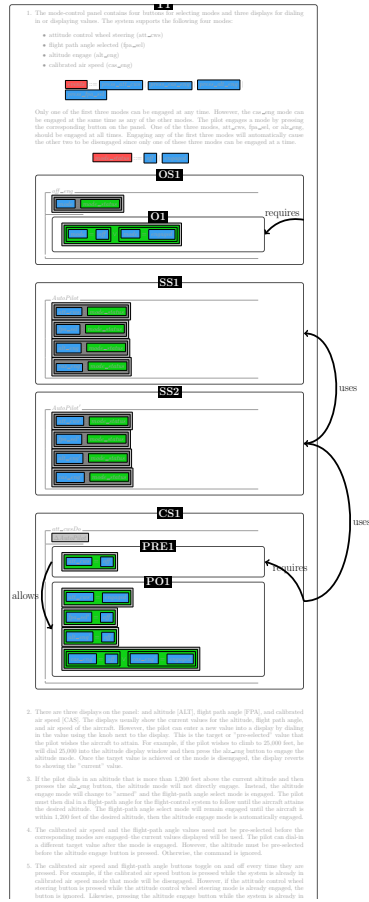


Figure 2.14: An example of the original Autopilot specification annotated in ZCGa and ZDRa.

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 correctly totalised. That is there is a precondition outstanding with not postcondition counter part. This does not matter for now 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 2.16 and 2.17):

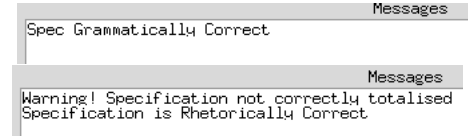


Figure 2.15: The outputting result when checking the autopilot specification with the ZCGa and ZDRa checkers.

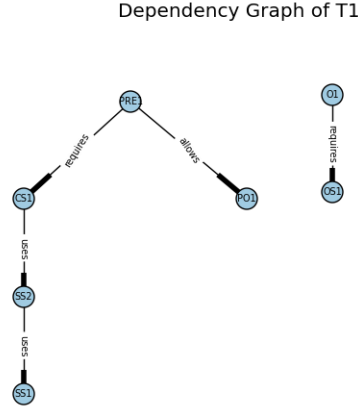


Figure 2.16: The dependency graph produced for the autopilot specification.

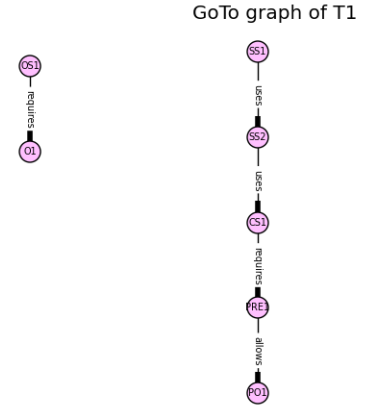


Figure 2.17: The goto graph produced for the autopilot specification.

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 separate as they do not have any relations with 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 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 dependent 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 exist.

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 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 consistency across the specification and thus need state invariants to be present.

```

theory gpsaln2
imports
Main

begin
(*DATATYPES*)

record SS1 =
(*DECLARATIONS*)

locale ln2 =
fixes (*GLOBAL DECLARATIONS*)
begin

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

definition CS1 ::
  "(*CS1_TYPES*) => bool"
where
  "CS1 (*CS1_VARIABLES*) ==
    (PRE1
    ^ (P01)"

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 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 \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 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 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:
  "( $\exists$  steamboiler0 :: SteamBoiler0.
 $\exists$  a' :: OnOff.
 $\exists$  steamboiler0' :: SteamBoiler0.
 $\exists$  w_max' :: nat.
 $\exists$  w_min' :: nat.
 $\exists$  z' :: State .
 $\exists$  v' :: OpenClosed.
  (z = norm)
 $\wedge$  (w < w_min  $\vee$ 
  w > w_max)
 $\wedge$  (a' = off  $\wedge$ 
  z' = stop)
 $\longrightarrow$  (w_min < w_max
 $\wedge$  (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.  $\exists$  steamboiler0 a' steamboiler0' w_max' w_min' z' v'.
      z = norm  $\wedge$  (w < w_min  $\vee$  w_max < w)  $\wedge$  a' = off  $\wedge$  z' = stop  $\longrightarrow$ 
      w_min < w_max  $\wedge$  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 though 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 automatically 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:
  "( $\exists$  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)
 $\wedge$  (m  $\in$  degModules)
 $\wedge$  ((p, m)  $\notin$  taking)
 $\wedge$  (taking' = taking  $\cup$  {(p, m)})
 $\wedge$  (students' = students)
 $\wedge$  (degModules' = degModules))
 $\wedge$  (Domain taking  $\subseteq$  students)
 $\wedge$  (Range taking  $\subseteq$  degModules)
 $\wedge$  (Domain taking'  $\subseteq$  students')
 $\wedge$  (Range taking'  $\subseteq$  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 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 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.  $\exists \text{degModules students taking } p \text{ degModules' students' taking' } m.$ 
    $(p \in \text{students} \wedge$ 
    $m \in \text{degModules} \wedge$ 
    $(p, m) \notin \text{taking} \wedge \text{taking}' = \text{taking} \cup \{(p, m)\} \wedge \text{students}' = \text{students} \wedge \text{degModules}' = \text{degModules}) \wedge$ 
    $\text{Domain taking} \subseteq \text{students} \wedge$ 
    $\text{Range taking} \subseteq \text{degModules} \wedge \text{Domain taking}' \subseteq \text{students}' \wedge \text{Range taking}' \subseteq \text{degModules}'$ 
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:
"( $\exists \text{degModules} :: \text{MODULE set}.$ 
 $\exists \text{students} :: \text{PERSON set}.$ 
 $\exists \text{taking} :: (\text{PERSON} * \text{MODULE}) \text{ set}.$ 
 $\exists p :: \text{PERSON}.$ 
 $\exists \text{degModules}' :: \text{MODULE set}.$ 
 $\exists \text{students}' :: \text{PERSON set}.$ 
 $\exists \text{taking}' :: (\text{PERSON} * \text{MODULE}) \text{ set}.$ 
 $\exists m :: \text{MODULE}.$ 
 $((p \in \text{students})$ 
 $\wedge (m \in \text{degModules})$ 
 $\wedge ((p, m) \notin \text{taking})$ 
 $\wedge (\text{taking}' = \text{taking} \cup \{(p, m)\})$ 
 $\wedge (\text{students}' = \text{students})$ 
 $\wedge (\text{degModules}' = \text{degModules})$ 
 $\wedge (\text{Domain taking} \subseteq \text{students})$ 
 $\wedge (\text{Range taking} \subseteq \text{degModules})$ 
 $\wedge (\text{Domain taking}' \subseteq \text{students}')$ 
 $\wedge (\text{Range taking}' \subseteq \text{degModules}'))"$ 
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

gendb and projectalloc, lemmas to prove for consitancy

2.4 Reflection and Discussion

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

2.4.2 Assumptions and limitations of the ZMathLang toolkit

Assumptions

- Specification = 1 theory. Can't do more than 1 specification in 1 document.
- we assume the user wishes to check for consistency. 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 doesn't 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] J.-R. Abrial. Event Based Sequential Program Development: Application to Constructing a Pointer Program. In K. Araki, S. Gnesi, and D. Mandrioli, editors, *FME*, volume 2805 of *Lecture Notes in Computer Science*, pages 51–74. Springer, 2003.
- [2] J.-R. Abrial. Formal methods in industry: achievements, problems, future. *Software Engineering, International Conference on*, 0:761–768, 2006.
- [3] M. Adams. Proof auditing formalised mathematics. *Journal of Formalized Reasoning*, 9(1):3–32, 2016.
- [4] A. Álvarez. *Automatic Track Gauge Changeover for Trains in Spain*. Vía Libre monographs. Vía Libre, 2010.
- [5] A. W. Appel. Foundational Proof-Carrying Code. In *LICS*, pages 247–256, 2001.
- [6] R. Arthan. Proof Power. <http://www.lemma-one.com/ProofPower/index/>, February 2011.
- [7] H. P. Barendregt. Lambda Calculi with Types. In *Handbook of Logic in Computer Science*, volume 2. Oxford University Press, 1991. <http://citeseer.ist.psu.edu/barendregt92lambda.html>Electronic Edition.
- [8] B. Beckert. An Example for Specification in Z: Steam Boiler Control. Universität Koblenz-Landau, Lecture Slides, 2004.

- [9] J. C. Blanchette. *Hammering Away, A user's guide to Sledgehammer for Isabelle/HOL*. Institut für Informatik, Technische Universität München, May 2015.
- [10] E. Borger and R. F. Stark. *Abstract State Machines: A Method for High-Level System Design and Analysis*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2003.
- [11] N. Bourbaki. *General topology. Chapters 1-4*. Elements of mathematics. Springer-Verlag, Berlin, Heidelberg, Paris, 1989. Trad. de : Topologie générale chapitres 1-4.
- [12] J. Bowen. Formal Methods Wiki, Z notation. http://formalmethods.wikia.com/wiki/Z_notation, July 2014.
- [13] A. D. Brucker, H. Hiss, and B. Wolff. HOL-Z 2.0: A Proof Environment for Z-Specifications. *Journal of Universal Computer Science*, 9(2):152–172, feb 2003.
- [14] L. Burski. Zmathlang. <http://www.macs.hw.ac.uk/~lb89/zmathlang/>, Jan 2016.
- [15] L. Burski. ZMathLang Website. <http://www.macs.hw.ac.uk/~lb89/zmathlang/examples>, June 2016.
- [16] R. W. Butler. An introduction to requirements capture using PVS: Specification of a simple autopilot. NASA Technical Memorandum 110255, NASA Langley Research Center, Hampton, VA, May 1996.
- [17] R. W. Butler. What is Formal Methods. <http://shemesh.larc.nasa.gov/fm/fm-what.html>, March 2001.
- [18] W. Chantatub. *The Integration of Software Specification Verification and Testing Techniques with Software Requirements and Design Processes*. PhD thesis, University of Sheffield, 1995.

- [19] Clearsy Systems Engineering. B Methode. <http://www.methode-b.com/en/>, 2013.
- [20] J. Coleman, C. Jones, I. Oliver, A. Romanovsky, and E. Troubitsyna. RODIN (rigorous open development environment for complex systems). In *EDCC-5, Budapest, Supplementary Volume*, pages 23–26, Apr. 2005.
- [21] I. E. Commission. IEC 61508 Functional Safety of Electrical/Electronic/Programmable Electronic Safety-related Systems. Technical report, International Electrotechnical Commission, 2010.
- [22] E. Currie. *The Essence of Z*. Prentice-Hall Essence of Computing Series. Prentice Hall Europe, 1999.
- [23] H. Curry. Functionality in combinatorial logic. In *Proceedings of National Academy of Sciences*, volume 20, pages 584–590, 1934.
- [24] C.Weidenbach, D.Dimova, A.Fietzke, R.Kumar, M.Suda, and P. Wischniewski. Isabelle cheat sheet. <http://www.phil.cmu.edu/~avigad/formal/FormalCheatSheet.pdf>.
- [25] C.Weidenbach, D.Dimova, A.Fietzke, R.Kumar, M.Suda, and P. Wischniewski. Spass. <http://www.spass-prover.org/publications/spass.pdf>.
- [26] N. de Bruijn. The mathematical vernacular, a language for mathematics with typed set. In *Workshop on Programming Logic*, 1987.
- [27] L. De Moura and N. Bjørner. Satisfiability Modulo Theories: Introduction and Applications. *Commun. ACM*, 54(9):69–77, Sept. 2011.
- [28] D. Fellar, F. Kamareddine, and L. Burski. Using MathLang to Check the Correctness of Specifications in Object-Z. In E. Venturino, H. M. Srivastava, M. Resch, V. Gupta, and V. Singh, editors, *In Modern Mathematical Methods and High Performance Computing in Science and Technology*, Ghaziabad, India, 2016. M3HPCST, Springer Proceedings in Mathematics and Statistics.

- [29] D. Feller. Using MathLang to check the correctness of specification in Object-Z. Master Thesis Report, 2015.
- [30] Formal Methods Europe, L-H Eriksson. Formal methods europe. http://www.fmeurope.org/?page_id=2, May 2016.
- [31] S. Fraser and R. Banach. Configurable Proof Obligations in the Frog Toolkit. In *Fifth IEEE International Conference on Software Engineering and Formal Methods (SEFM 2007), 10-14 September 2007, London, England, UK*, pages 361–370. IEEE Computer Society, 2007.
- [32] J. Groote, A. Osaiweran, and Wesselius2. Benefits of Applying Formal Methods to Industrial Control Software. Technical report, Eindhoven University of Technology, 2011.
- [33] S. L. Hantler and J. C. King. An Introduction to Proving the Correctness of Programs. *ACM Comput. Surv.*, 8(3):331–353, Sept. 1976.
- [34] E. C. R. Hehner. Specifications, Programs, and Total Correctness. *Sci. Comput. Program.*, 34(3):191–205, 1999.
- [35] A. Ireland. Rigorous Methods for Software Engineering, High Integrity Software Intensive Systems. Heriot Watt Universtiy, MACS, Lecture Slides.
- [36] F. Kamareddine and J.B.Wells. A research proposal to UK funding body. Formath, 2000.
- [37] F. Kamareddine, R. Lamar, M. Maarek, and J. B. Wells. Restoring Natural Language as a Computerised Mathematics Input Method. In M. Kauers, M. Kerber, R. Miner, and W. Windsteiger, editors, *Calculementus/MKM*, volume 4573 of *Lecture Notes in Computer Science*, pages 280–295. Springer, 2007.
- [38] F. Kamareddine, M. Maarek, K. Retel, and J. B. Wells. Gradual computerisation/formalisation of mathematical texts into Mizar. In *From Insight to Proof: Festschrift in Honour of Andrzej Trybulec*, pages 81–95. Springer-Verlag, 2007.

- [39] F. Kamareddine, M. Maarek, and J. B. Wells. Toward an Object-Oriented Structure for Mathematical Text. In M. Kohlhase, editor, *MKM*, volume 3863 of *Lecture Notes in Computer Science*, pages 217–233. Springer, 2005.
- [40] F. Kamareddine and R. Nederpelt. A refinement of de Bruijn’s formal language of mathematics. *Logic, Language and Information*, 13(3):287–340, 2004.
- [41] F. Kamareddine, J. B. Wells, and C. Zengler. Computerising mathematical texts in MathLang. Technical report, Heriot-Watt University, 2008.
- [42] Khosrow-Pour and Mehdi, editors. *Encyclopedia of Information Science and Technology*. IGI Global,, 2 edition.
- [43] S. King, J. Hammond, R. Chapman, and A. Pryor. Is Proof More Cost-Effective Than Testing? *IEEE Trans. Software Eng.*, 26(8):675–686, 2000.
- [44] Kolyang, T. Santen, and B. Wolff. *Theorem Proving in Higher Order Logics: 9th International Conference, TPHOLs’96 Turku, Finland, August 26–30, 1996 Proceedings*, chapter A structure preserving encoding of Z in isabelle/HOL, pages 283–298. Springer Berlin Heidelberg, Berlin, Heidelberg, 1996.
- [45] Kolyang, T. Santen, B. Wolff, R. Chaussee, I. Gmbh, and D.-S. Augustin. Towards a Structure Preserving Encoding of Z in HOL, 1986.
- [46] A. Krauss. Defining Recursive Functions in Isabelle/HOL , 2008.
- [47] R. Lamar. The MathLang Formalisation Path into Isabelle – A Second-Year report, 2003.
- [48] R. Lamar. *A Partial Translation Path from MathLang to Isabelle*. PhD thesis, Heriot-Watt University, 2011.
- [49] R. Lamar, F. Kamareddine, and J. B. Wells. MathLang Translation to Isabelle Syntax. In J. Carette, L. Dixon, C. S. Coen, and S. M. Watt, editors, *Calculus/MKM*, volume 5625 of *Lecture Notes in Computer Science*, pages 373–388. Springer, 2009.

- [50] P. G. Larsen, N. Battle, M. Ferreira, J. Fitzgerald, K. Lausdahl, and M. Verhoef. The overture initiative integrating tools for vdm. *SIGSOFT Softw. Eng. Notes*, 35(1):1–6, Jan. 2010.
- [51] I. Lee, J. Y.-T. Leung, and S. H. Son. *Handbook of Real-Time and Embedded Systems*. Chapman & Hall/CRC, 1 edition, 2007.
- [52] K. R. M. Leino. Dafny: An Automatic Program Verifier for Functional Correctness. In E. M. Clarke and A. Voronkov, editors, *LPAR (Dakar)*, Lecture Notes in Computer Science, pages 348–370. Springer, 2010.
- [53] M. Lindgren, C. Norström, A. Wall, and R. Land. Importance of Software Architecture during Release Planning. In *WICSA*, pages 253–256. IEEE Computer Society, 2008.
- [54] I. E. U. Ltd and H. . S. Laboratory. A methodology for the assignment of safety integrity levels (SILs) to safety-related control functions implemented by safety-related electrical, electronic and programmable electronic control systems of machines. Standard, Health and Safety Executive (HSE), Mar. 2004.
- [55] M. Maarek. Mathematical documents faithfully computerised: the grammatical and text & symbol aspects of the MathLang framework, First Year Report, 2003.
- [56] M. Maarek. *Mathematical documents faithfully computerised: the grammatical and test & symbol aspects of the MathLang Framework*. PhD thesis, Heriot-Watt University, 2007.
- [57] M. Mahajan. Proof Carrying Code. *INFOCOMP Journal of Computer Science*, 6(4):01–06, 2007.
- [58] M. Mihaylova. ZMathLang User Interface Internship Report. Internship Report, 2015.
- [59] M. Mihaylova. ZMathLang User Interface User Manual. Intern User Manual, 2015.

- [60] G. C. Necula and P. L. 0001. Safe, Untrusted Agents Using Proof-Carrying Code. In G. Vigna, editor, *Mobile Agents and Security*, volume 1419 of *Lecture Notes in Computer Science*, pages 61–91. Springer, 1998.
- [61] I. C. Office. International Electrotechnical Commission. <http://www.iec.ch/>, July 2016.
- [62] S. Owre, S. Rajan, J. Rushby, N. Shankar, and M. Srivas. PVS: combining specification, proof checking, and model checking. In R. Alur and T. A. Henzinger, editors, *Computer-Aided Verification, CAV '96*, number 1102 in *Lecture Notes in Computer Science*, pages 411–414, New Brunswick, NJ, July/August 1996. Springer-Verlag.
- [63] R. L. Page. Engineering Software Correctness. *J. Funct. Program.*, 17(6):675–686, 2007.
- [64] B. C. Pierce. *Types and Programming Languages*. MIT Press, Cambridge, MA, USA, 2002.
- [65] W. R. Plugge and M. N. Perry. American Airlines' "Sabre" Electronic Reservations System. In *Papers Presented at the May 9-11, 1961, Western Joint IRE-AIEE-ACM Computer Conference*, IRE-AIEE-ACM '61 (Western), pages 593–602, New York, NY, USA, 1961. ACM.
- [66] K. Retel. *Gradual Computerisation and Verification of Mathematics: MathLang's Path into Mizar*. PhD thesis, Heriot-Watt University, 2009.
- [67] A. Riazanov and A. Voronkov. The design and implementation of vampire. *Journal of AI Communications*, 15(2/3):91–110, 2002.
- [68] G. Rossum. Python Reference Manual. Technical report, Python Software Foundation, Amsterdam, The Netherlands, The Netherlands, 1995.
- [69] M. Saaltink and O. Canada. The Z/EVES 2.0 User's Guide, 1999.
- [70] S. Schulz. E—a brainiac theorem prover. *Journal of AI Communications*, 15(2/3):111–126, 2002.

- [71] J. M. Spivey. *The Z Notation: A Reference Manual*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1989.
- [72] M. Spivey. Z Reference Card. <https://spivey.oriel.ox.ac.uk/mike/fuzz/refcard.pdf>. Accessed on November 2014.
- [73] M. Spivey. Towards a Formal Semantics for the Z Notation. Technical Report PRG41, OUCL, October 1984.
- [74] M. Spivey. The fuzz manual. *Computing Science Consultancy*, 34, 1992.
- [75] S. Stepney. A tale of two proofs. In *BCS-FACS third Northern formal methods workshop, Ilkley*, 1998.
- [76] I. UK. *Customer Information Control System (CICS) Application Programmer's Reference Manual*. White Plains, New York.
- [77] University of Cambridge and Technische Universitat Munchen. Isabelle. <http://www.isabelle.in.tum.de>, May 2015.
- [78] Z. Wen, H. Miao, and H. Zeng. Generating Proof Obligation to Verify Object-Z Specification. In *Proceedings of the International Conference on Software Engineering Advances (ICSEA 2006), October 28 - November 2, 2006, Papeete, Tahiti, French Polynesia*, page 38. IEEE Computer Society, 2006.
- [79] A. Whitehead and B. Russell. *Principia Mathematica*. Number v. 2 in Principia Mathematica. University Press, 1912.
- [80] J. Woodcock and A. Cavalcanti. A tutorial introduction to designs in unifying theories of programming. In *Integrated Formal Methods*, pages 40–66. Springer, 2004.
- [81] J. Woodcock and J. Davies. *Using Z: Specification, Refinement, and Proof*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1996.
- [82] C. Zengler. MathLang- Towards a Better Usability and Building the Path into Coq, First Year Report. Technical report, Heriot-Watt University, November 2008.