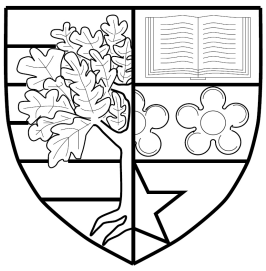
FROM FORMAL SPECIFICATION TO FULL PROOF: A STEPWISE METHOD

by

Lavinia Burski



Submitted for the degree of
Doctor of Philosophy

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

March 2016

Abstract

Write an abstract

Acknowledgements

Contents

1 From ZDRa to General Proof Sketch **1**

1.1 What is a General Proof Sketch 1

1.2 Creating the Graph 2

1.3 Proof Obligations 5

1.3.1 POb1, Proof Obligation type 1 5

1.3.2 POb2, Proof Obligation type 2 6

1.3.3 Proof Obligations in the General Proof Skeleton 7

1.3.3.1 Proof Obligations in specification examples 10

1.4 Conclusion 10

Bibliography **12**

Todo list

Write an abstract 2

Find year for atelier b proof obligation user manual 11

Chapter 1

From ZDRa to General Proof Sketch

The skeletons described in this chapter are automatically generated if the specification passes the Z Document Rhetorical aspect (ZDRa) check. Section 1.1 describes the general proof skeleton. Which uses the graphs generated in the ZDRa to provide the order the instaces should go to into any theorem prover. Section **??** then explains how a general proof skeleton can be automatically translated into a skeleton in Isabelle format automatically. In section **??** we describe how the Isabelle Skeleton can be used to fully prove a formal specification which requires two steps, the first is an automatic step to fill in the Isabelle skeleton and the final step is up to the user to prove the lemma's and properties of the specification.

1.1 What is a General Proof Sketch

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

We then run an algorithm on the GoTo graph to generate a proof skeleton. Figure 1.1 shows part of the code in generating this proof sketch.

- *allnodes*, is a set of all the instances labelled by the user of a specification in ZDRa.

- *fromnodes*, is a set containing all the nodes which are dependent on another instance.

- *tonodes*, is a set containing all the nodes which have some other nodes dependent on them.

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

Figure 1.1: Part of the algorithm to create a proof sketch.

1.2 Creating the Graph

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

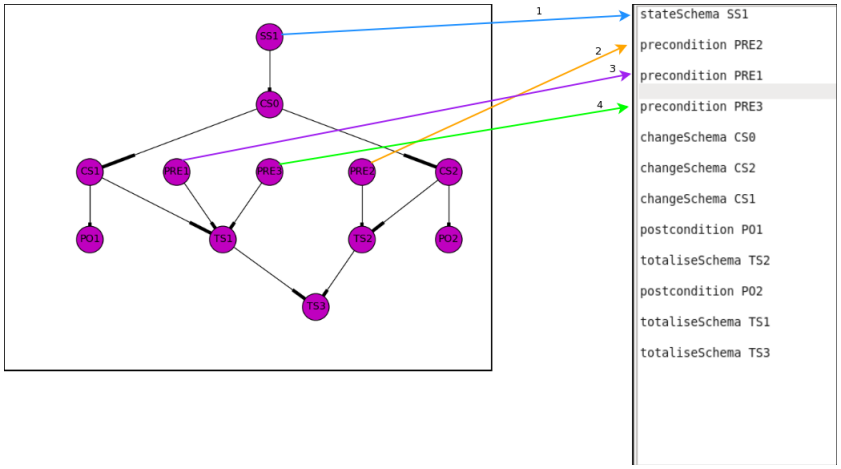


Figure 1.2: GoTo graph and proof skeleton of vending machine step 1.

First of all the program looks at all the nodes of the GoTo graph and prints out all the nodes which are not dependent on anything. That is, they may have

successors but they have no predecessors, they do not use or need anything else and can stand by themselves. These nodes can be printed in any order, so in diagram 1.2 we see that we have SS1, PRE1 PRE2 and PRE3 all printed.

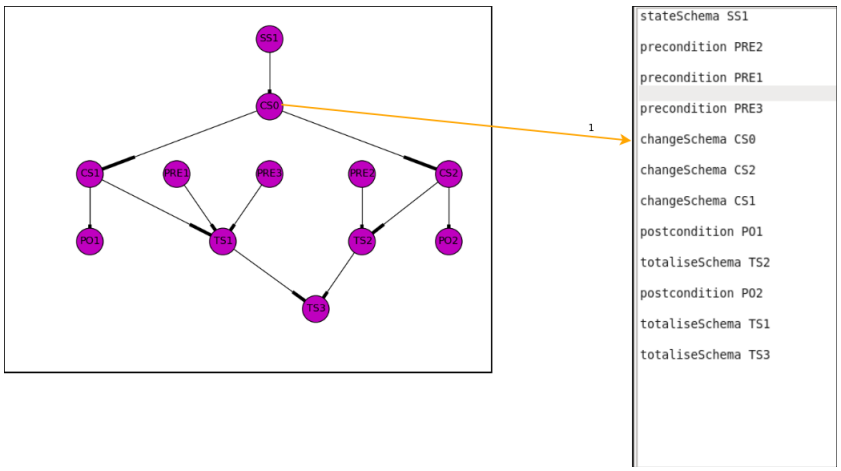


Figure 1.3: GoTo graph and proof skeleton of vending machine step 2.

The next part of the algorithm checks whether there exists a node in the GoTo graph where all of its parents are printed out in the proof skeleton. Figure 1.3 shows that the next node to be in the proof skeleton is CS0.

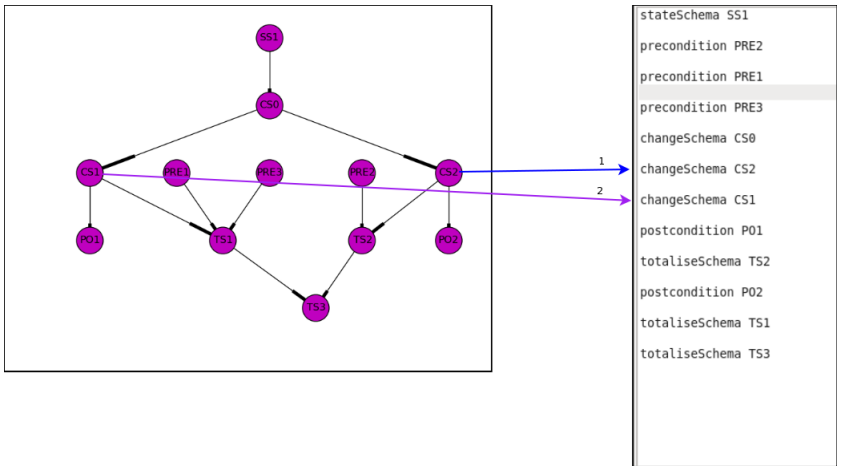


Figure 1.4: GoTo graph and proof skeleton of vending machine step 3.

The next part we see that after CS0 is added to the proof skeleton then both CS1, and CS2 can be added. This is shown in figure 1.4.

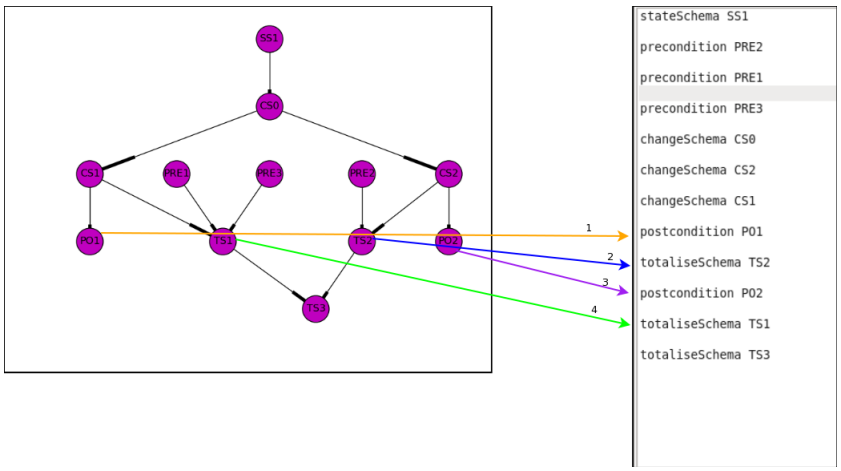


Figure 1.5: GoTo graph and proof skeleton of vending machine step 4.

Figure 1.5 shows the next stage of adding nodes to the Proof Skeleton. Since CS1 and CS2 are now added to the proof skeleton then the next row of nodes can be added. Since PO1 only had one parent (CS1) it is added first, PO2 also had one parent (CS2) it is added second. The others had more parents which are already in the proof sketch so they are added next randomly.

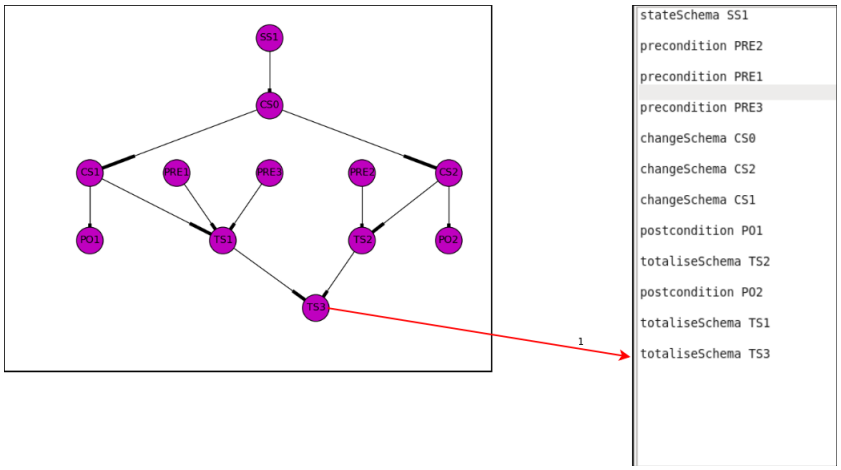


Figure 1.6: GoTo graph and proof skeleton of vending machine step 5.

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

1.3 Proof Obligations

There are many properties one may wish to prove about their specification. These certain properties are called proof obligations. Proof obligations for formal notations are an entire research area in their own right. However as the MathLang framework for Z specifications (ZMathLang) framework concentrates on giving the novice an idea of how to prove their specification we will focus on checking the specification for consistancy. Using the description in [63], checking the specification is consitant can fall under two categories:

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

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

1.3.1 POb1, Proof Obligation type 1

Definition 1.3.1. *POb1*

$$Context \vdash \exists\, var, var' \bullet PRE\# \land PO\# \longrightarrow SI\# \land SI\#'$$

where var and var' are the variables and variables' used in the schema with their types, SI# is the state Invariants of the specification, SI#' is the state invariants prime in the specification, PRE# is the precondition of the schema, PO# is the post operation of the schema and # is some arbitary number.

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

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

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

The ZDRa syntax of this proof obligation would be :

```
lemma AddStudentDoesntChangeSI:
" \<exists> (*CS1_variables :: CS1_TYPES*).
(PRE1)
\<and> (PO1)
\<longrightarrow> ((SI1)
\<and (SI1'))"
```

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

1.3.2 POb2, Proof Obligation type 2

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

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

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

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

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

1.3.3 Proof Obligations in the General Proof Skeleton

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

```
330 def createdProofObligations(someGraph):
331     listofProofObligations = []
332     i = 0
333     setofCSandOS = []
334     lemmaSet = []
335     #This part takes the existing lemmas and finds the highest number
336     #Eg if there are lemmas (L1, L2, TS4, L5) it will return 5 being
337     #the maximum lemma
338     for a, b in someGraph:
339         if b == "lemma":
340             lemmaSet.append(int(filter(str.isdigit, a)))
341     if lemmaSet:
342         i = max(lemmaSet)
343     else:
344         i = 0
345     #This part finds all operational and change Schemas so we can check the
346     #state invariants are satisfied
347     for a, b in someGraph:
348         if b == "changeSchema":
349             setofCSandOS.append((a,b))
350     if setofCSandOS:
351        # print i
352         for zdraName, zdraType in setofCSandOS:
353             indexOfZdra = setofCSandOS.index((zdraName, zdraType))
354             #NumberOfNewLemma is the highest number of existing
355             #lemmas plus 1
356             NumberOfNewLemma = indexOfZdra + i + 1
357             LemmaAndNumber = "L"+ str(NumberOfNewLemma)+"_("+ zdraName+ ")"
358             listofProofObligations.append((LemmaAndNumber, "lemma"))
359     return listofProofObligations
```

Figure 1.7: Part of the algorithm to create Proof Obligation ZDRa names.

The proof obligations which check that changeSchema's and outputSchema's follow the stateInvariants are added to the original general proof skeleton. The general proof skeleton starts of with being an ordered list (GPSaOL), then the algorithm for generating a list of proof obligations is run and added to the original proof skeleton. Figure 1.7 shows the algorithm which creates the ordered list of proof obligations.

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

For Example if we had the following GpsaOL:

[(SS1, stateSchema), (IS1, initialSchema), (CS1, changeSchema), (CS2, changeSchema), (TS1, totaliseSchema)]

Then in this case:

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

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

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

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

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

Figure 1.8: Example of a General Proof Skeleton with lemma's.

An example of a General Proof Skeleton with added proof obligations is shown in Figure 1.8. The changeSchema's in this specification are CS1 and CS2. Therefore to make sure the changeSchemas do not change the state of the specification and comply with the state invariants the two lemma's L1_CS1 and L2_CS2 have been added.

1.3.3.1 Proof Obligations in specification examples

Since the vending machine specification (appendix **??**) doesn't have any stateInvaritants then the General Proof Skeleton aspect (Gpsa) will not have any added proof obligations to check for consistance. That is we can't check that the postconditions do not change the state if the state has no restrictions. However the birthdaybook example (appendix **??**) does have stateInvariants. Therfore we must add properties to check that any changeSchema's follow the state restrictions. Part of the birthdaybook Gpsa is shown in figure 1.9. Since there are stateInvariants (SI1) and a changing state Schema (CS1) then the proof obligation `L1_(CS1)` has been added to the Gpsa.

```
stateSchema SS1
stateInvarients SI1
initialSchema IS1
postcondition PO2
outputSchema OS1
precondition PRE2
changeSchema CS1
totaliseSchema TS1
.......
lemma L1_(CS1)
```

Figure 1.9: Part of the GPSa for the birthdayBook example. (Full version shown in appendix **??**)

1.4 Conclusion

This chapter describes how the ZDRa program uses the GoTo graph to generate a general proof skeleton (step 2→3 in figure **??**). The general proof skeleton is an automatically generated .txt file which displays the order in which this instances must go in a theorem prover to be logically correct. This chapter gives a basic

understanding of proof obligations for Z and examples which proof obligations are automatically generated when translating Z specifications into Isabelle. We give a formal definition of the proof obligation to check for consistancy of the state invariants and show an example. The next chapter describes how the general proof skeleton is translated into Isabelle syntax.

Chapter 2

General Proof Sketch aspect and beyond

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

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

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

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

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

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

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

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

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

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

Then when translating into the Isabelle skeleton it becomes:

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

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

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

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

This would translate to the Isabelle skeleton as:

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

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

2.1 Proof Obligations in Isabelle Syntax

Lemmas which are proof obligations. That is instances with the a ZDRa name `L#_CS#` where '`#`' is a number become `lemma's` in Isabelle syntax. The translation from the General Proof Skeleton aspect (GPSa) into Isabelle syntax depends if the changeSchema in question has a precondition, postcondition or both. We use definition 1.3.1 in aid with the translation.

2.1.1 Proof Obligation translation where the schema has a precondition

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

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

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

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

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

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

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

2.1.2 Proof Obligation translation where the changeSchema has only postcondition

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

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

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

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

2.2 Benefits

Refer back to appendix of semiform

The Isabelle skeleton allows for incomplete specifications to also be automated into a half-baked proof. This way the user can have a general outline for their specification and fill in the missing information directly into the Isabelle skeleton at a later stage. For this reason it is good to have an Isabelle skeleton before the full filled in half-baked proof. The user then has an outline if they wish to add to the specification directly as they will have an example of how the instances should be translated.

2.3 ZCGa specification to Fill in the Isabelle Skeleton

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

It is important to note that all the Z Core Grammatical aspect (ZCGa) annotations at this stage disappear as the labelled information is automatically put into Isabelle syntax.

2.3.1 Z Types and FreeTypes

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

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

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

If the specification had the following freetype:

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

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

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

is added to the skeleton.

2.3.2 Declarations

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

```
\draschema{OS1}{
\begin{schema}{ab}
d: \power COLOUR
c: COLOUR
\where
\draline{PO1}{c \in d}
\end{schema}}
```

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

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

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

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

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

2.3.3 Expressions

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

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

Syntax in Z	**Syntax in LaTeX**	**Syntax in Isabelle**
{...}	\{...\}	{...}
(| ... |)	\limg...\rimg	\<lparr>...\<rparr>
⟨...⟩	\langle...\rangle	\<langle>...\<rangle>
# A	\#	card *if A is set*, length *if A is list*
∪	\cup	\<union>
∩	\cap	\<inter>
×	\cross	\<times>
\	\setminus	-

	$\geq$	\geq	\<ge>
	$\leq$	\leq	\<le>
	$\lhd$	\lhd	\<lhd>
	$\rhd$	\rhd	\<rhd>
	$\rhd$	\nrres	\<unlhd>
	$\lhd$	\ndres	\<unrhd>
	$\Rightarrow$	\implies	\<Longrightarrow>
	$\Leftrightarrow$	\iff	\<Longleftrightarrow>
	$\notin$	\notin	\<notin>
	$\in$	\in	\<in>
$\subset$	\subset		\<subset>
$\subseteq$	\subseteq		\<subseteq>
$\land$	\land		\<and>
$\lor$	\lor		\<or>
$\neg$	\lnot		\<not>
$\neq$	\neq		\<noteq>
$a \mapsto b$	a \mapsto b		(a,b)
		length *if set preceding using*	\<rightharpoonup>
$\mathbb{P}\, A$	\power A		A set
$\mathbb{N}$	\nat		nat
$\mathbb{N}_1$	\nat_1		nat
$\mathbb{Z}$	\num		num
$A \pfun B$	A \pfun B		(A \<rightharpoonup> B)
$A \fun B$	A \fun B		(A * B) set
$A \rel B$	A \rel B		(A * B) set
seq A	\seq A		A list
iseq A	\iseq A		A list
$\text{seq}_1\, A$	\iseq_1 A		A list
dom A	\dom A		Domain A
		dom *if set preceding using*	\<rightharpoonup>
ran A	\ran A		Range A
		ran *if set preceding using*	\<rightharpoonup>
$\exists$	\exists		\<exists>
$\forall$	\forall		\<forall>
$\bullet$	@		.
$R^\sim$	R\inv		\<R^-1>
R^k	R^{k}		\<R^k>

Table 2.1: A table showing the symbols which are changed from Z specifications in LaTeX to Isabelle.

Another part of the Z mathematical toolkit which we need to rewrite are the use of partial functions. In HOL all functions are total but there are ways to do certain proofs about partial functions [37]. Therefore the variables which have a type as a partial function will are translated as a set of pairs. Any proofs to check for partial functions if needed can be done by the user in step 6 shown in figure **??** but the details of these proofs are beyond the scope of this thesis.

2.3.4 Schema Names

The Names of the Schema are added to the skeleton by using the ZDRa name. For example if the specification had the line `\draschema{TS1}{\begin{schema}{ab}{..` then anywhere '`TS1`' is listed in the skeleton it will be converted to '`ab`'. This is done throughout the entire skeleton.

2.3.5 Proof Obligations

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

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

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

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

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

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

This lemma filled in becomes the following proof obligation:

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

2.4 Filled in Isabelle Skeleton to a Full Proof

The final step to get from a half-baked proof into a full proof is labelled as step 6 in Figure **??**, this is also named fill in 2. Technically the specification the user automatically generates in fill in 1 is fully formalised in Isabelle if there are no other properties to be proved. If the specification is not fully formalised, using the half-baked proof generated in step 5, the user then adds any safety properties

about the specification they wish to prove in the form of *lemmas*. As the properties will be specific to the user and/or specification it is difficult to automate this step. Therefore some theorem prover knowledge may be required for step 6. Some of the automated theorem prover tools such as Sledgehammer [9] may be useful when proving the properties.

Figure **??** shows an example of a proof obligations generated by ZMathLang proved in Isabelle. Again we have highlighted the user input in red. To help prove these properties wed have used sledgehammer [9] which is part of the Isabelle/Hol package. The full proof of this specification can be found in appendix **??**. At this point, proving properties in a theorem prover may require some expertise in the field. Proving tools such as Satisfiability Modulo Theories (SMT) solvers [19] such as sledgehammer are an interesting area of research on it's own however these such details are out with the scope of this thesis.

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


by blast
```

Figure 2.1: An example of a proof completed by user input.

2.5 Conclusion

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

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

Find year for atelier b proof obligation user manual

Bibliography

[1] HOL-Z 2.0: A Proof Environment for Z-Specifications. *Journal of Universal Computer Science*, 9(2):152–172, Feb. 2003.

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

[3] 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.

[4] J.-R. Abrial. Formal methods in industry: achievements, problems, future. *Software Engineering, International Conference on*, 0:761–768, 2006.

[5] A. Álvarez. *Automatic Track Gauge Changeover for Trains in Spain*. Vía Libre monographs. Vía Libre, 2010.

[6] A. W. Appel. Foundational Proof-Carrying Code. In *LICS*, pages 247–256, 2001.

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

[8] H. P. Barendregt. Lambda Calculi with Types. In *Handbook of Logic in Computer Science*, volume 2. Oxford University Press, 1991. http://citeseer.ist.psu.edu/barendregt92lambda.htmlElectronic Edition.

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

[10] J. Bowen. Formal Methods Wiki, Z notation, July 2014. Accessed:01/07/2014.

[11] L. Burski. Zmathlang. `http://www.macs.hw.ac.uk/~lb89/zmathlang/`, Jan 2016.

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

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

[14] W. Chantatub. *The Integration of Software Specification Verification and Testing Techniques with Software Requirements and Design Processes.* PhD thesis, University of Sheffield, 1995.

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

[16] E. Currie. *The Essence of Z.* Prentice-Hall Essence of Computing Series. Prentice Hall Europe, 1999.

[17] H. Curry. Functionality in combinatorial logic. In *Proceedings of National Academy of Sciences*, volume 20, pages 584–590, 1934.

[18] N. de Bruijn. The mathematical vernacular, a language for mathematics with typed set. *In Workshop on Programming Logic*, 1987.

[19] L. De Moura and N. Bjørner. Satisfiability Modulo Theories: Introduction and Applications. *Commun. ACM*, 54(9):69–77, Sept. 2011.

[20] 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.

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

[22] Formal Methods Europe, L-H Eriksson. Formal methods europe. `http://www.fmeurope.org/?page_id=2`, May 2016.

[23] 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.

[24] J. Groote, A. Osaiweran, and Wesselius2. Benefits of Applying Formal Methods to Industrial Control Software. Technical report, Eindhoven University of Technology, 2011.

[25] S. L. Hantler and J. C. King. An Introduction to Proving the Correctness of Programs. *ACM Comput. Surv.*, 8(3):331–353, Sept. 1976.

[26] E. C. R. Hehner. Specifications, Programs, and Total Correctness. *Sci. Comput. Program.*, 34(3):191–205, 1999.

[27] A. Ireland. Rigorous Methods for Software Engineering, High Integrity Software Intensive Systems. Heriot Watt Universtiy, MACS, Lecture Slides.

[28] F. Kamareddine and J.B.Wells. A research proposal to UK funding body. Formath, 2000.

[29] F. Kamareddine, R. Lamar, M. Maarek, and J. B. Wells. Restoring Natural Language as a Computerised Mathematics Input Method. In M. Kauers, M. Kerber, R. Miner, and W. Windsteiger, editors, *Calculemus/MKM*, volume 4573 of *Lecture Notes in Computer Science*, pages 280–295. Springer, 2007.

[30] 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.

[31] 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.

[32] F. Kamareddine and R. Nederpelt. A refinement of de Bruijn's formal language of mathematics. *Logic, Language and Information*, 13(3):287–340, 2004.

[33] F. Kamareddine, J. B. Wells, and C. Zengler. Computerising mathematical texts in MathLang. Technical report, Heriot-Watt University, 2008.

[34] 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.

[35] 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.

[36] Kolyang, T. Santen, B. Wolff, R. Chaussee, I. Gmbh, and D.-S. Augustin. Towards a Structure Preserving Encoding of Z in HOL, 1986.

[37] A. Krauss. Defining Recursive Functions in Isabelle/HOL , 2008.

[38] R. Lamar. The MathLang Formalisation Path into Isabelle – A Second-Year report, 2003.

[39] R. Lamar. *A Partial Translation Path from MathLang to Isabelle*. PhD thesis, Heriot-Watt University, 2011.

[40] R. Lamar, F. Kamareddine, and J. B. Wells. MathLang Translation to Isabelle Syntax. In J. Carette, L. Dixon, C. S. Coen, and S. M. Watt, editors, *Calculemus/MKM*, volume 5625 of *Lecture Notes in Computer Science*, pages 373–388. Springer, 2009.

[41] 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.

[42] M. Lindgren, C. Norstrm, A. Wall, and R. Land. Importance of Software Architecture during Release Planning. In *WICSA*, pages 253–256. IEEE Computer Society, 2008.

[43] M. Maarek. Mathematical documents faithfully computerised:the grammatical and text & symbol aspects of the MathLang framework, First Year Report, 2003.

[44] M. Maarek. *Mathematical documents faithfully computerised: the grammatical and test & symbol aspects of the MathLang Framework*. PhD thesis, Heriot-Watt University, 2007.

[45] M. Mahajan. Proof Carrying Code. *INFOCOMP Journal of Computer Science*, 6(4):01–06, 2007.

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

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

[48] 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.

[49] S. Owre, S. Rajan, J. M. Rushby, N. Shankar, and M. Srivas. PVS: Combining Specification, Proof Checking, and Model Checking. pages 411–414. Springer-Verlag, 1996.

[50] R. L. Page. Engineering Software Correctness. *J. Funct. Program.*, 17(6):675–686, 2007.

[51] B. C. Pierce. *Types and Programming Languages*. MIT Press, Cambridge, MA, USA, 2002.

[52] 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.

[53] K. Retel. *Gradual Computerisation and Verification of Mathematics: Math-Lang's Path into Mizar*. PhD thesis, Heriot-Watt University, 2009.

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

[55] M. Saaltink and O. Canada. The Z/EVES 2.0 User's Guide, 1999.

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

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

[58] M. Spivey. Towards a Formal Semantics for the Z Notation. Technical Report PRG41, OUCL, October 1984.

[59] M. Spivey. The fuzz manual. *Computing Science Consultancy*, 34, 1992.

[60] S. Stepney. A tale of two proofs. In *BCS-FACS third Northern formal methods workshop, Ilkley*, 1998.

[61] I. UK. *Customer Information Control System (CICS) Application Programmer's Reference Manual*. White Plains, New York.

[62] University of Cambridge and Technische Universitat Munchen. Isabelle. `http://www.isabelle.in.tum.de`, May 2015.

[63] 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.

[64] J. Woodcock and A. Cavalcanti. A tutorial introduction to designs in unifying theories of programming. In *Integrated Formal Methods*, pages 40–66. Springer, 2004.

[65] J. Woodcock and J. Davies. *Using Z: Specification, Refinement, and Proof*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1996.