

Using MathLang to Check the Correctness of Specifications in Object-Z

David Feller, Fairouz Kamareddine, and Lavinia Burski

Heriot Watt University, Edinburgh

Abstract. The importance of thoroughly checking software specifications is widely recognised in the software industry, particularly for software involved in dealing with safety critical systems. The MathLang project has been successfully used to check large mathematical texts for correctness in a stepwise fashion. Currently MathLang is being tested for checking the correctness of formal specifications written in Z. Since object-orientation is a vital concept for software specification, it is important that the tools available for thoroughly checking specifications can be used with a language powerful enough to express specifications for object oriented software. This paper aims to test the usefulness of MathLang for the computerisation of formal specifications written in Object-Z.

1 Introduction

Inadequate checking of software is a serious problem in the software industry. According to Frenieu [5]:

Experience shows that more than 75% of finished software products have errors during maintenance, and deadlines are missed and cost overruns are a rule not an exception. It was estimated that more than 50% of the development effort was spent on testing and debugging. Nevertheless, some errors are not detected by testing, and some of them are never detected. More, there are projects that have never been finished. And it is not an exception; it is estimated that from each six large projects two of them are never finished.

Rigorous checking of software systems could help with these issues. This is of particular importance to the designers of safety critical systems who cannot afford to find bugs in their software by testing it on users. This is because such a bug could cause injury or death in the course of being found. According to MacKenzie [11], the total number of computer-related accidental deaths, worldwide, to the end of 1992, can be expressed, in conventional format, as 1,100 +/- 1,000. One can easily imagine other cases where a high degree of confidence that a piece of software will always function as intended is needed before it can be used (e.g. software dealing with sensitive information). This paper is concerned with creating software which aids in the formal proof of software correctness.

1.1 Why Formally Prove Software Correctness?

Formally proving that a piece of software is correct can give us a high degree of confidence that it will function as intended; checking the validity of that proof,

even more so. Testing and debugging can fail because of some condition that the software developers forgot to check. As Dijkstra [4] observed: program testing can be used to show the presence of bugs, but never to show their absence. Testing and debugging can take much longer than expected to locate those errors that are hard to isolate and hence fixing those errors can only happen late in the development cycle. However, a specification that has been proven correct should function as defined, provided that said proof is correct.

1.2 The Difficulties of Formally Proving Correctness

For large systems, formally proving correctness can be repetitive and labour intensive. It is not guaranteed that software developers have a great deal of experience with formal proof and it is certainly not guaranteed that every developer working on a large piece of software could aid in the formal checking of software correctness. As [12] states, many software engineers reject the use of formal methods for software validation, arguing that it is too complex and time-consuming a process for most programmers. Further it is still possible for such proofs to be subject to human error. As such, it is important that we have good tools to aid in formally checking the correctness of software specifications.

Contributions This paper presents the first step in the development of a new tool to aid in a *stepwise easy to use fashion* in the formal checking of the correctness of software specifications written in the specification language Object Z [10, 16]. This first step allows for the type checking and grammatical correctness of documents written in Object Z. We present a development path for expansion of the tool to aid in more complete checking of specifications in Object Z where also logical and rhetorical correctness can be checked. We also explain why our proposed method might provide a basis for development of tools for checking correctness in other specification languages.

1.3 Related Work

One usually formalises a Z [6] specification into a complete proof right away [17, 18, 1, 2], as shown by arrow **e** in Figure 1. The thickness of the arrow here represents the level of difficulty, the huge expertise needed, and the amount of work necessary to take that path. Our proposal is to carry out the correctness checking in smaller steps, each of which is more focused and very simple to carry out. These smaller steps are based on MathLang [7]. MathLang is a system for computerising mathematical texts which aims to reduce the complexity of checking the correctness of a text by breaking down the process into more manageable steps which can be easily completed with the aid of a computer. MathLang starts by separating out the work that needs to be done in computerising a mathematical text into three main aspects. These are the Core Grammatical aspect (CGa), the Document Rhetorical aspect (DRa) and the Text and Symbol aspect (TSa).

The CGa checks the internal grammatical structure of a text is correct by capturing the structures and common concepts with a finite set of concepts

which are derived from weak type theory. The TSa captures the mathematical relations which hold between the parts of the text as represented by the CGa. The DRa captures the logical roles that are held by chunks of text that tell us where they feature in an argument or proof. This information can be used to generate a proof skeleton in a theorem prover making the move from document to formal checking much simpler. Further it allows simple checking of the grammar and general structure of the document to be performed automatically.

MathLang for Z (ZMathLang) [3], divides **e** into a number of smaller paths via **a**, **b**, **c** and **d**. Following this path the user would apply the Z core grammatical aspect (ZCGa) and the Z document rhetorical aspect (ZDRa), to show that the specification is weakly type checked and is also rhetorically correct (i.e. no loops in the reasoning). Then the user would take the ZCGa and ZDRa annotated specification into a general proof skeleton, then a ‘half-baked’ proof and ultimately a complete proof. Breaking this down would allow a user with minimal theorem prover expertise to obtain a fully proved Z specification.

Unlike MathLang for mathematics, ZMathLang does not require a Text-Symbolic aspect (TSa) as the mathematical relations in Z are already formal.

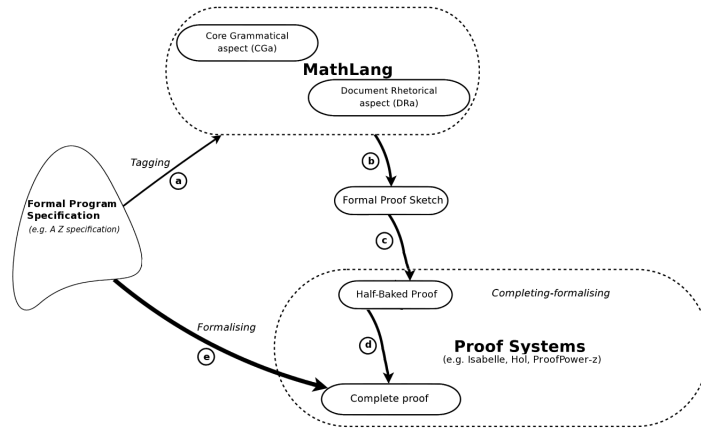


Fig. 1. The different steps taken to achieve a full proof using the ZMathLang method.

Object-Z Object-Z [13] is an extension of the Z language for writing formal specifications that has added functionality for dealing with object oriented concepts. Both Z and Object-Z have been designed to make formally proving the correctness of specifications relatively easy. They each have a standard notation which can be easily manipulated in a mathematical fashion, allowing for proof of correctness using standard mathematical methods.

Z allows specifications to be split up into different schema boxes - each of which represent individual functions within the software. The input, output and

manipulation of data is expressed through a mathematical notation based around set theory. Object Z introduces class boxes, which allow a specification to be identified with a class of objects and standard notation for creating instances of objects and for initialising and running methods within a specification.

Why choose Object-Z over Z as a Specification Language for Correctness Checking? Object oriented programming allows developers to separate out programs into modules whose functions and interactions are (relatively) easy for developers to understand and whose contents are easier to alter without needing to change too much of the rest of the code. This is especially important for large pieces of software with multiple developers whose code can quickly grow unmanageable and difficult for humans to interpret. Most popular languages in use today are object oriented. Examples include Java, Python, C++, Visual Basic .NET and Ruby. It is important, therefore, that our tool for checking formal correctness works for specifications of object oriented languages. Smith [13] noted the following benefits of an object oriented specification language:

- The modularity it brings to system design. Modularity increases the clarity of specifications by allowing a reader to focus on one part at a time.
- It provides a precise methodology for system design. This methodology involves the specification of a system by first specifying the behaviour of its constituent objects by classes, and by utilizing inheritance and polymorphism where appropriate
- Seamless development - the use of common concepts and system structuring at each stage of system development: from the specification right through to the implementation. This is possible when using an object-oriented approach to specification and then implementing in an object-oriented programming language. It makes the specification more accessible to the programmer, who may not be a formalist, and facilitates his or her task of transforming the specification to implementation.

Another reason to prefer Object-Z over Z when using MathLang to check the correctness of formal specifications is that checking the correctness of Object-Z gives us much more confidence that MathLang is well suited to checking the correctness of software specifications than checking the correctness of Z alone does, as object oriented models might present unique problems for conversion into a framework like that of MathLang due to the subtyping of objects.

Tools for Object-Z TOZE [14] is a graphical editor for Object-Z documents which allows syntax and type checking without demanding experience with LaTeX or requiring the user to save the specification and use tools outside the text editor. Kimber [8] gives a tool which maps 80% of Object-Z to perfect developer [15] allowing the direct verification of the soundness of simple specifications.

By checking Object-Z in MathLang, one allows some flexibility in whether to perform syntax and weak type checking, or check that simple dependencies are fulfilled, or provide a full proof. Another benefit especially when the soundness of

an Object-Z specification is difficult to verify directly, consists of the MathLang automated layers which are crucial to embed the entire specification into a theorem prover. This could particularly be useful in large software projects where no or very few individuals are familiar with a theorem prover, and it would take a long time to translate the full specification into a theorem prover by hand.

1.4 Overview

In section 2 we create a new weak typing system for MathLang to incorporate Object-Z specifications. We give the weak types and the rules in which we check specifications. In section 2.2 we give a step by step example of how we can check for weak typing errors in an object-Z specification. We explain how to label the specification and how to run the weak type checker on the specification. Section 3 goes on to explain how we implemented the Object-Z Core Grammatical aspect (OZCGa), and how it has been tested. The problems encountered and how they were dealt with is described in section 3.4. Finally, our conclusion along with benefits and limitations is described in section 4.

2 Adapting MathLang to include Object-Z

We look at the first aspect of the ZMathLang framework (CGa) extended to weak type check Object-Z specifications.

OZCGa includes 7 weak types **Spec**, **Γ** , **\mathcal{T}** , **\mathcal{S}** , **\mathcal{Z}** , **\mathcal{E}** , **\mathcal{D}** , **\mathcal{O}** , **\mathcal{M}** corresponding to `specification`, `schematext`, `term`, `set`, `declaration`, `expression`, `definition`, `object` and `method` respectively. We categorise the parts of the Z syntax using these types in order to define the core grammatical aspect OZCGa.

We have three types of variables in our syntax:

- $V = V^{\mathcal{T}}$, Variables giving terms.
- $V = V^{\mathcal{S}}$, Variables giving sets.
- $V = V^{\mathcal{O}}$, Variables giving objects.

We have three types of constants in our syntax:

- $C = C^{\mathcal{T}}$, Constants for terms.
- $C = C^{\mathcal{S}}$, Constants for sets.
- $C^{\mathcal{E}}$, Constants giving expressions. These can be further broken down into the following:
 - C^{bool} , constants $\subseteq, =, \neq$ taking expressions as parameters.
 - C^{termop} , constant term operators $<, \leq, =, >, \geq, \neq$ taking two terms as parameters.
 - C^{setop} , constant set operators $\subseteq, =, \neq, partition$ taking a set and a sequence of sets as parameters.
 - $C^{termsetop}$, constant term/set operators \in, \notin taking one term and one set as parameters.
 - C^{objop} , constant object operators $=, \neq$ taking two objects as parameters.

- $C^{objsetop}$, constant object/set operators \in, \notin taking one object and one set as parameters.

We have three types of constants for our Object-Z syntax:

- $C = C^{\circledast}$, Constants for objects.
- $C = C^{\mathcal{M}}$, Constants for methods.
 - $C = C^{obop}$, constant for object operator taking an object as a parameter.
 - $C = C^{methop}$, constant for method operator \circledast taking two methods as parameters.

We have three types of binders in Z:

- $B^{\mathbb{S}}$, the binder \cup giving sets and taking sets as parameters.
- $B^{\mathcal{E}}$, binders \exists, \forall giving expressions and taking expressions as parameters.
- \downarrow , gives an object and takes an object as its parameter.

Definitions in Object-Z can define constants including those of the form C^{\circledast} which take a specification as its parameter.

Declarations express the relationship between something and its type. In the ZCGa we have two kinds of declarations, these can be *SET* (the type of all sets) or a particular set. We write either $V^{\mathbb{S}} : SET$, $V^{\mathcal{T}} : \mathbb{S}$ or $V^{\circledast} : \mathbb{S}$.

Expressions, terms and sets in Z are given as described in our rules for constants variables and binders.

A schematext can be empty or it can contain a declaration, expression or method. A declaration in a schematext represents the introduction of a new variable of a known type.

A specification is either empty or it consists of schematext and definitions where the parts of the schematext which are not defined inside the schematext itself have a corresponding definition in the specification.

2.1 A Formalization of these Typing Rules

If we formally represent these typing rules we see that they are a subset of the typing rules of MathLang. The only differences [3] are that we change book to specification, context becomes schematext and statements become expressions. We eliminate nouns and adjectives and only have one syntax for definition.

We use the notation $::$ for typing between an entity and its weak type and \vdash to denote derivability. Here are some examples (we only state the meaning of the first 3 and leave the rest as obvious):

1. spec is a weakly typed specification:
 $\vdash \text{spec} :: \mathbf{Spec}$
2. Γ is a weakly well typed paragraph relative to specification spec:
 $\text{spec} \vdash \Gamma :: \mathbf{\Gamma}$
3. t is a weakly typed term, relative to specification spec and schematext Γ :
 $\text{spec}; \Gamma \vdash t :: \mathcal{T}$

4. $\text{spec}; \Gamma \vdash s :: \mathbb{S}$
5. $\text{spec}; \Gamma \vdash Z :: \mathcal{Z}$
6. $\text{spec}; \Gamma \vdash e :: \mathcal{E}$
7. $\text{spec}; \Gamma \vdash D :: \mathcal{D}$
8. $\text{spec}; \Gamma \vdash o :: \mathbb{O}$
9. $\text{spec}; \Gamma \vdash m :: \mathcal{M}$

The next definition is crucial for analysing the grammatical correctness of the specification since it collects the defined constants and declared variables of specifications and paragraphs:

Definition 1.

1. Let $\theta \in \text{spec}$ be a definition paragraph $\Gamma \triangleleft D$ where D is of the form $c(x_1, \dots, x_n) := A$. We define $\text{defcons}(D) = c$.
2. $\text{defcons}(\text{spec}) = \{ \text{defcons}(D) \mid \Gamma \triangleleft D \text{ is a paragraph of spec for some } \Gamma \}$.
3. Internal constants defined using $==$ are noted as $\text{defcons}(\Gamma)$.
4. We define for parameter P the weak type of P with respect to spec and Γ as: $\text{wt}_{\text{spec}; \Gamma}(P) = W$ if and only if $\text{spec}; \Gamma \vdash P :: W$.
5. Predefined constants such as \mathbb{P} , dom and ran are noted as $\text{prefcons}(\text{spec})$.
6. We use $\text{OK}(\text{spec}; \Gamma)$ to denote $\vdash \text{spec} :: \mathbf{Spec}$ and $\text{spec} \vdash \Gamma :: \mathbf{\Gamma}$.
7. We define $\text{dvar}(\Gamma)$ as follows:
 - (a) if $\Gamma = \emptyset$, then $\text{dvar}(\Gamma) = \emptyset$.
 - (b) if $\Gamma = \Gamma', x : A$ and $x \notin \text{dvar}(\Gamma')$, then $\text{dvar}(\Gamma) = \text{dvar}(\Gamma') \cup x$.
 - (c) Otherwise, if $\Gamma = \Gamma', e$, then $\text{dvar}(\Gamma) = \text{dvar}(\Gamma')$.

The next definition gives the typing rules that deal with type-orientedness:

Definition 2.

1. Derivation rule for variables:

$$\frac{\text{OK}(\text{spec}, \Gamma), x \in V^{\mathcal{T}/\mathbb{S}/\mathbb{O}}, x \in \text{dvar}(\Gamma)}{\text{spec}; \Gamma \vdash x :: \mathcal{T}/\mathbb{S}/\mathbb{O}} (\text{var})$$

2. Derivation rule for internal constants:

$$\frac{\text{OK}(\text{spec}, \Gamma), \Gamma \triangleleft D \in \text{spec}, \text{dvar}(\Gamma') = (x_1, \dots, x_n), \text{defcons}(D) = c \in C^{\mathcal{T}/\mathbb{S}/\mathbb{O}/\mathcal{E}/\mathcal{M}}, \text{wt}_{\text{spec}; \Gamma}(P_i) = \text{wt}_{\text{spec}; \Gamma'}(x_i), \text{for all } i=1, \dots, n}{\text{spec}; \Gamma \vdash c(P_1, \dots, P_n) :: \mathcal{T}/\mathbb{S}/\mathbb{O}/\mathcal{E}/\mathcal{M}} (\text{int} - \text{cons})$$

3. Derivation rule for external constants:

$$\frac{\text{OK}(\text{spec}, \Gamma), c \text{ external to spec}, c :: k_1, \dots, k_n \rightarrow k, \text{spec}; \Gamma \vdash P_i :: k_i (i=1, \dots, n)}{\text{spec}; \Gamma \vdash c(P_1, \dots, P_n) :: k} (\text{ext} - \text{cons})$$

4. Derivation rule for binders:

$$\frac{\text{OK}(\text{spec}; \Gamma; Z), b \in B, b :: k_1 \rightarrow k_2, \text{spec}; \Gamma, Z \vdash E :: k_1}{\text{spec}; \Gamma \vdash b_z(E) :: k_2} (\text{bind})$$

5. Derivation rule for definitions:

$$\frac{\text{spec}, \Gamma \vdash o :: \mathbb{O}, \text{OK}(\text{spec}', \Gamma'), \\ c \in C^{\mathbb{O}}, c \notin \text{prefcons}(\text{spec}) \cup \text{defcons}(\text{spec})}{\text{spec}; \Gamma \vdash \text{classbox}(c(\text{spec}', \Gamma'), o) :: D} (\text{obj} - \text{def})$$

That is, an entity that consists of a set equalling a number of variables bound by a constant and that has not yet been defined has weak type definition.

6. Derivation rule for an empty schematext is:

$$\frac{\vdash \text{spec} :: \mathbf{Spec}}{\text{spec} \vdash \emptyset :: \mathbf{\Gamma}} (\text{emp} - \text{cont})$$

7. Derivation rule for adding a set declaration to a paragraph is:

$$\frac{\text{OK}(\text{spec}; \Gamma), x \in V^{\mathbb{S}}, x \notin \text{dvar}(\Gamma)}{\text{spec} \vdash \Gamma, x : \text{SET} :: \mathbf{\Gamma}} (\text{set} - \text{dec})$$

8. Derivation rule for adding a term declaration is:

$$\frac{\text{OK}(\text{spec}; \Gamma), \text{spec}; \Gamma \vdash s :: \mathbb{S}, x \in V^{\mathbb{T}}, x \notin \text{dvar}(\Gamma)}{\text{spec} \vdash \Gamma, x : s :: \mathbf{\Gamma}} (\text{term} - \text{dec})$$

9. Derivation rule for adding an object declaration to a paragraph:

$$\frac{\text{OK}(\text{spec}; \Gamma), \text{spec}; \Gamma \vdash s :: \mathbb{S}/\mathbb{O}, x \in V^{\mathbb{O}}, x \notin \text{dvar}(\Gamma)}{\text{spec} \vdash \Gamma, x : s :: \mathbf{\Gamma}} (\text{obj} - \text{dec})$$

10. Derivation rule for adding an expression is:

$$\frac{\text{OK}(\text{spec}; \Gamma), \text{spec}; \Gamma \vdash e :: \mathcal{E}}{\text{spec} \vdash \Gamma, x : e :: \mathbf{\Gamma}} (\text{assump})$$

11. Derivation rule for adding methods:

$$\frac{\text{OK}(\text{spec}; \Gamma), \text{spec}; \Gamma \vdash m :: \mathcal{M}}{\text{spec} \vdash \Gamma, x : m :: \mathbf{\Gamma}} (\text{meth})$$

12. Derivation rule for an empty specification:

$$\frac{}{\vdash \emptyset :: \mathbf{Spec}} (\text{emp} - \text{spec})$$

13. Derivation rule for extending a specification is:

$$\frac{\text{spec} \vdash \Gamma :: \mathbf{\Gamma}}{\vdash \text{spec}, \Gamma :: \mathbf{Spec}} (\text{spec} - \text{ext})$$

2.2 An Example of an Object-Z Class With Weak Types Labelled

We have implemented the above syntax and type derivation rules to obtain an OZCGa type checker that checks whether specifications written in Object-Z are grammatically correct. To use this type checker, we need first to annotate the Object-Z specification with our weak types. For this, we create commands within a \LaTeX package (see Table 1) where each of the weak types, is associated to a \LaTeX command and the colour in which the contents appears.










Weak Type	Command	Colour
specification	$\backslash\text{specification}\{\dots\}$	
schematext	$\backslash\text{text}\{\dots\}$	
term	$\backslash\text{term}\{\dots\}$	
set	$\backslash\text{set}\{\dots\}$	
declaration	$\backslash\text{declaration}\{\dots\}$	
expression	$\backslash\text{expression}\{\dots\}$	
definition	$\backslash\text{definition}\{\dots\}$	
object	$\backslash\text{object}\{\dots\}$	
method	$\backslash\text{method}\{\dots\}$	

Table 1. The \LaTeX commands to annotate an Object-Z specification with.

We use an example of an Object-Z specification ‘TwoCards’ which describes an action where the balance is the first card plus the second card. Money is allowed to be withdrawn on both these cards. An example of the specification is shown in Figure 5 (source file in Figure 2). We use the commands from Table 1 to annotate this specification, giving us the source code shown in Figure 3.

Colours now appear around each grammatical part of the specification (Figure 5). These colours can be used to reduce the the complexity of the specification and can also assist beginners in learning the syntax of Object-Z. More examples of the labelling and weakly typed Object-Z specification can be found in Appendix D. Other examples of weakly typed Z specifications are also given in Appendixes A, B and C.

After the specification has been labelled using the ‘ozcga’ package in \LaTeX , our weak type checker goes through the specification and checks it for grammatical correctness. This weak type checking is run in a terminal through a program implemented in python.

3 Implementation of the OZCGa

In this section we look at the specific implementation of the Core Grammatical aspect of MathLang for Object-Z (OZCGa), that we created. We go over the implementation and the specific examples we used to test it.

```

\begin{schema}{TwoCards}
c1,c2:CreditCard \
totalbal:\num
\where
c1 \neq c2\
totalbal = c1.balance + c2.balance
\end{schema}

\begin{schema}{withdraw1}
\where
c1.withdraw
\end{schema}

\begin{schema}{withdraw2}
\where
c2.withdraw
\end{schema}

\begin{schema}{transferAvail}
\where
c1.withdrawAvail \semi c2.deposit
\end{schema}

```

Fig. 2. Part of an Object-Z specification source code.

```

\begin{class}{\object{TwoCards}}
\also
\specification{
\begin{schema}{TwoCards}
\text{\declaration{\object{c1},\object{c2}:
\expression{CreditCard}}\
\declaration{\term{totalbal}:\expression{\num}}}}
\where
\text{\expression{\object{c1}\neq\object{c2}}\
\expression{\term{totalbal}=
\term{\object{c1}.\term{balance}+\object{c2}.
\term{balance}}}}
\end{schema}
\begin{schema}{withdraw1}
\where
\text{\method{\object{c1}.withdraw}}
\end{schema}
\begin{schema}{withdraw2}
\where
\text{\method{\object{c2}.withdraw}}
\end{schema}
\begin{schema}{transferAvail}
\where
\text{\method{\method{
\object{c1}.withdrawAvail}\semi\method{\object{c2}.
deposit}}}}
\end{schema}}
\end{class}

```

Fig. 3. Part of an Object-Z specification labelled in OZCGa source code. The full version can be found in E.

3.1 Expansion of Existing Functional Software

The \LaTeX style file used for labelling Object-Z specifications to be checked by the OZCGa.py is built around the oz style file [19], which is the file the Community Z Tools website [21] suggests for typesetting Object-Z documents in \LaTeX , so it can be easily applied to existing Object-Z documents. The software for the OZCGa itself is built around the software used for the ZCGa - so we can be fairly confident that the functionality and reliability of the ZCGa has been preserved in the adaptation to the OZCGa. In addition it means that Z specifications which can be checked by the ZCGa do not need changing to be checked with the OZCGa.

3.2 The style file

The zcga.sty style file used for labelling Z specifications for the ZCGa imports the zed.sty style file. For the ozcga.sty style file we chose to import the oz.sty style file to deal with specifications written in Object-Z. We chose oz.sty because it contained all the commands used in the zed.sty style file so ZCGa specifications written in \LaTeX with the ozcga.sty (see Appendixapp:stylefile) style file in place of the zcga.sty style file are compatible with the ZCGa. We have also kept the original weak type labels and used the same format for labelling the two additional weak types object and method.

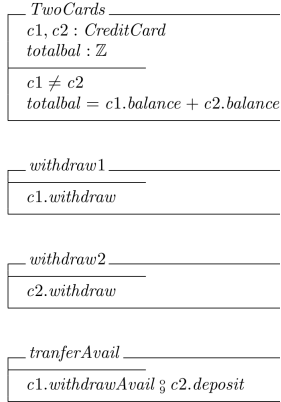


Fig. 4. Part of an Object-Z specification.

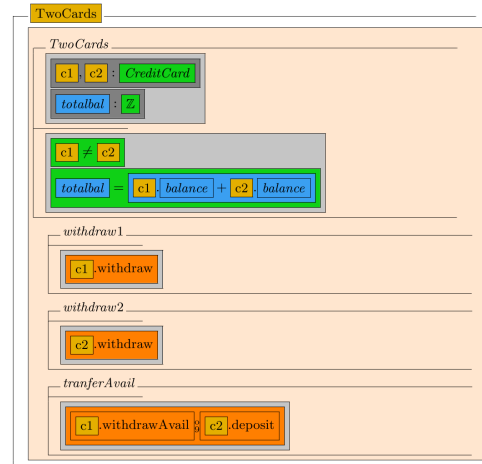


Fig. 5. Part of an Object-Z specification labelled in OZCGa and compiled with pdflatex. The full version can be found in Appendix D

3.3 The Code Structure

We have retained all the original code for the ZCGa python file (though we have added to several functions within it as well as incorporating new functions). This makes us much more confident in the backwards compatibility of the OZCGa. It also gives us a very high degree of confidence that the structures present in both Z and Object-Z are type checked correctly.

Where possible we have copied the parts of each typing rule for terms as closely as possible when expanding the rules to deal with objects. Most the expression constants between objects and sets and between objects and other objects are the same as those between terms and sets and those between terms and other terms respectively. Similarly the declaration rule for objects closely resembles the declaration rules for terms. This allows us to have a high degree of confidence that these rules are well implemented - as they follow the same structure as well implemented code.

3.4 Problems Encountered

While the typing rules of Z are perfectly compatible with the object definition rule of Object-Z the way in which they are realised in the zcga.py file makes the object declaration rule difficult to implement without extensive restructuring of the code. The reason for this is that the ZCGa assumes that only one specification is checked at a time so the checking of specifications is not separable from the checking of documents. There are two problems with this approach when

checking Object-Z specifications which require that the specifications contained within them are themselves well typed:

- A document can easily contain two specifications which are individually badly typed, but together well typed (if types only defined in one specification are used in the other).
- A document can easily contain two well typed specifications which would be badly typed if treated like they were one specification (if the same type is declared in each specification).

Our Solution Since we wanted to retain as much of the structure of the ZCGa as possible our solution is to ask the user to break down their specification so that only one class is checked at once. In order to do this all data from other classes, apart from the declarations of terms called within the class being checked using `object.term` should be deleted. An example of how this breakdown is done can be found in Appendix E.

4 Conclusion

In this paper we described a new translation path from an Object-Z specification into a theorem prover in a step-wise fashion. We have derived and implemented a new weak typing system for Object Z and thus completing the first step in the ZMathLang path. This weak typing system is defined by weak types and derivation rules for object-orientedness and shows that the system can weakly type check Object-Z specifications. The style file which was produced also acts as a clear visual referenece for the typing of an Object-Z specification, many errors can be caught at the stage of adding these labels.

This paper is concerned with the Core Grammatical aspect of Object-Z. The next step is taking Object-Z specification through to the Document Rhetorical aspect to check the document rhetorical correctness (e.g. loops in the reasoning). Using the DRa we can automatically produce dependencies graphs and the proof skeleton. Other work which might be of interest is to create the MathLang path for other languages such as SysML or for specifications which are written non-formally in natural language.

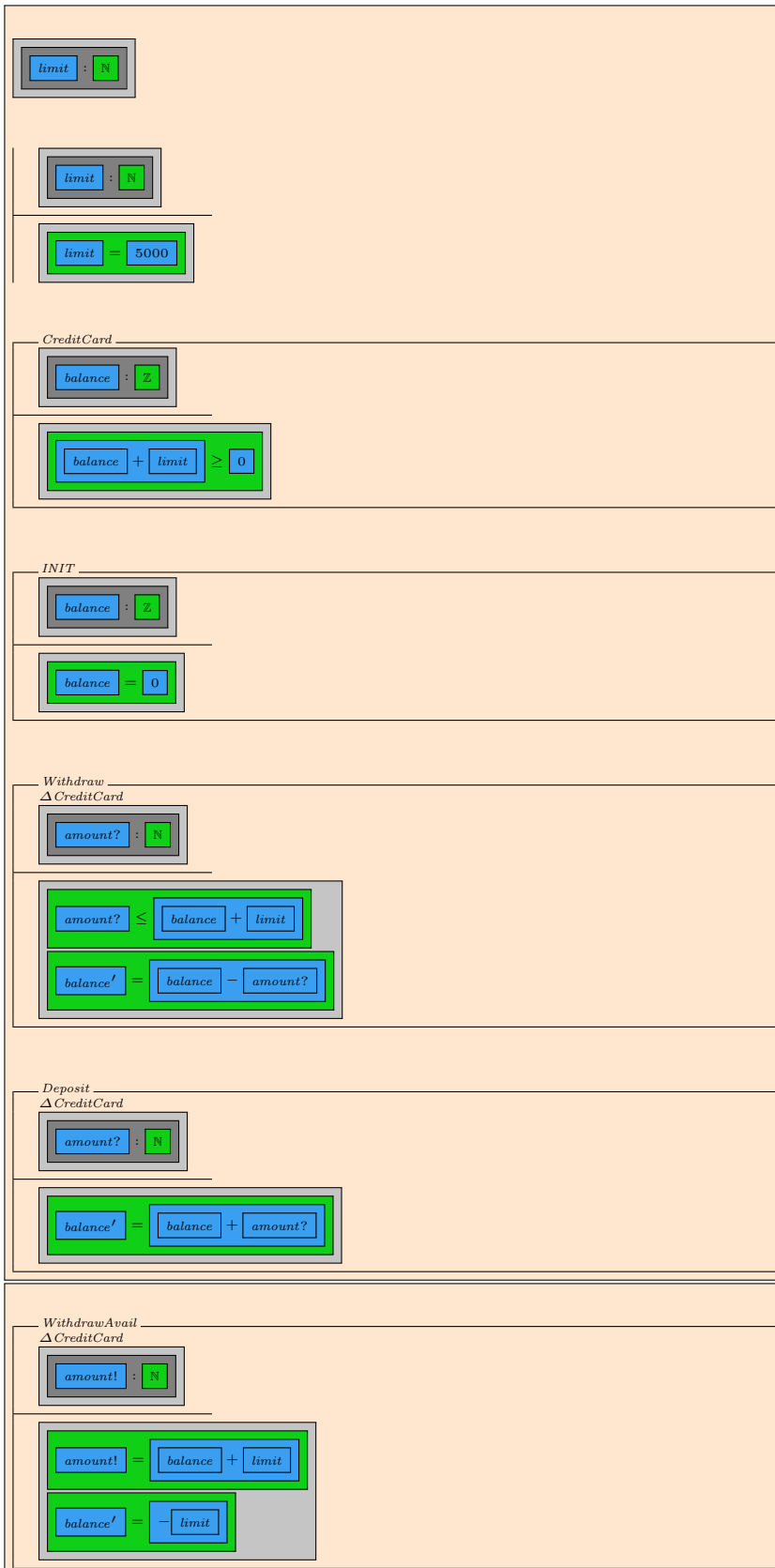
Limitations to the ZDRa is that specifications need to be labelled by hand and the program runs on a terminal. Perhaps having a user interface to label the specification and run the program may make things more friendly to use.

References

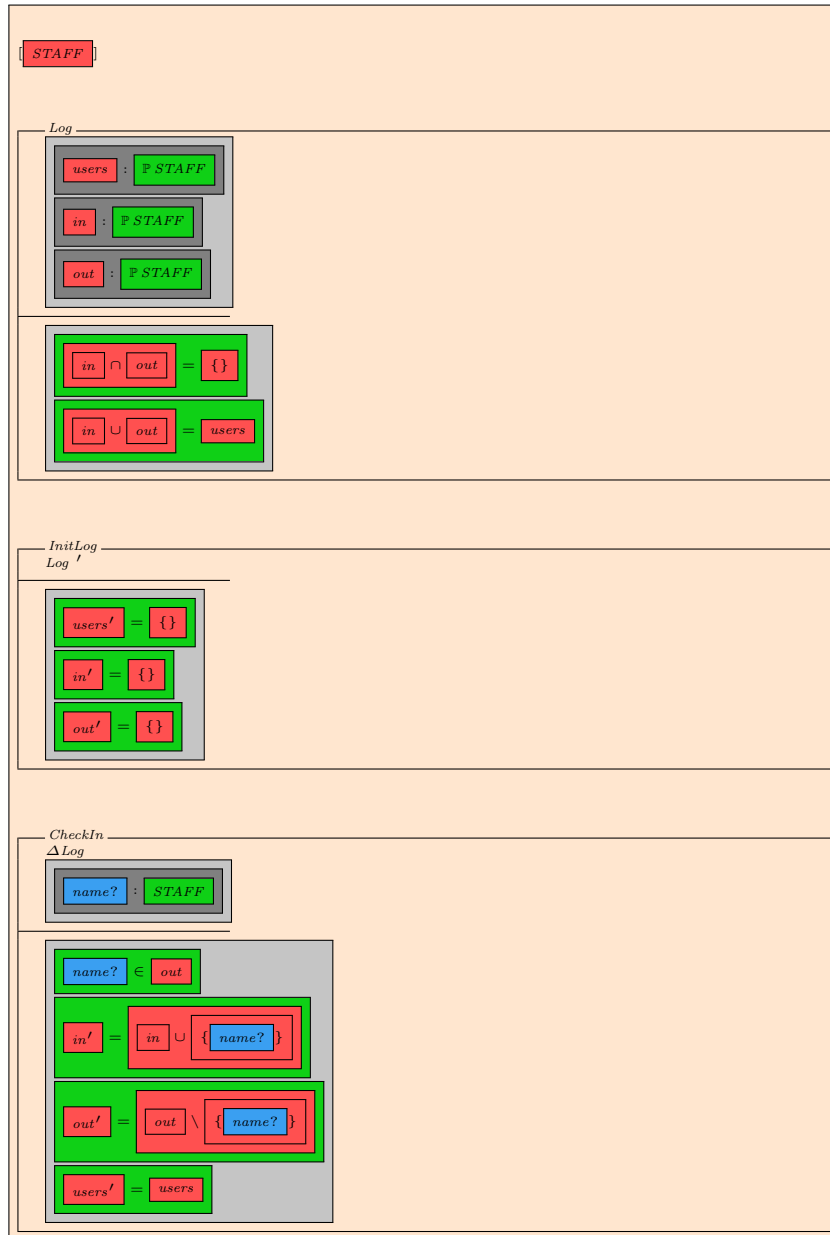
1. Brucker, A. D., Rittinger, F. and Wolff, B., HOL-Z 2.0: A Proof Environment for Z-Specifications. *Journal of Universal Computer Science*, 9(2), 152–172, 2003.
2. <https://www.brucker.ch/projects/hol-z/> (Accessed April 2015).
3. Burski, L. and Kamareddine, F., Translating Z into Isabelle Syntax using MathLang. ULTRA Group, Heriot Watt University, 2015.

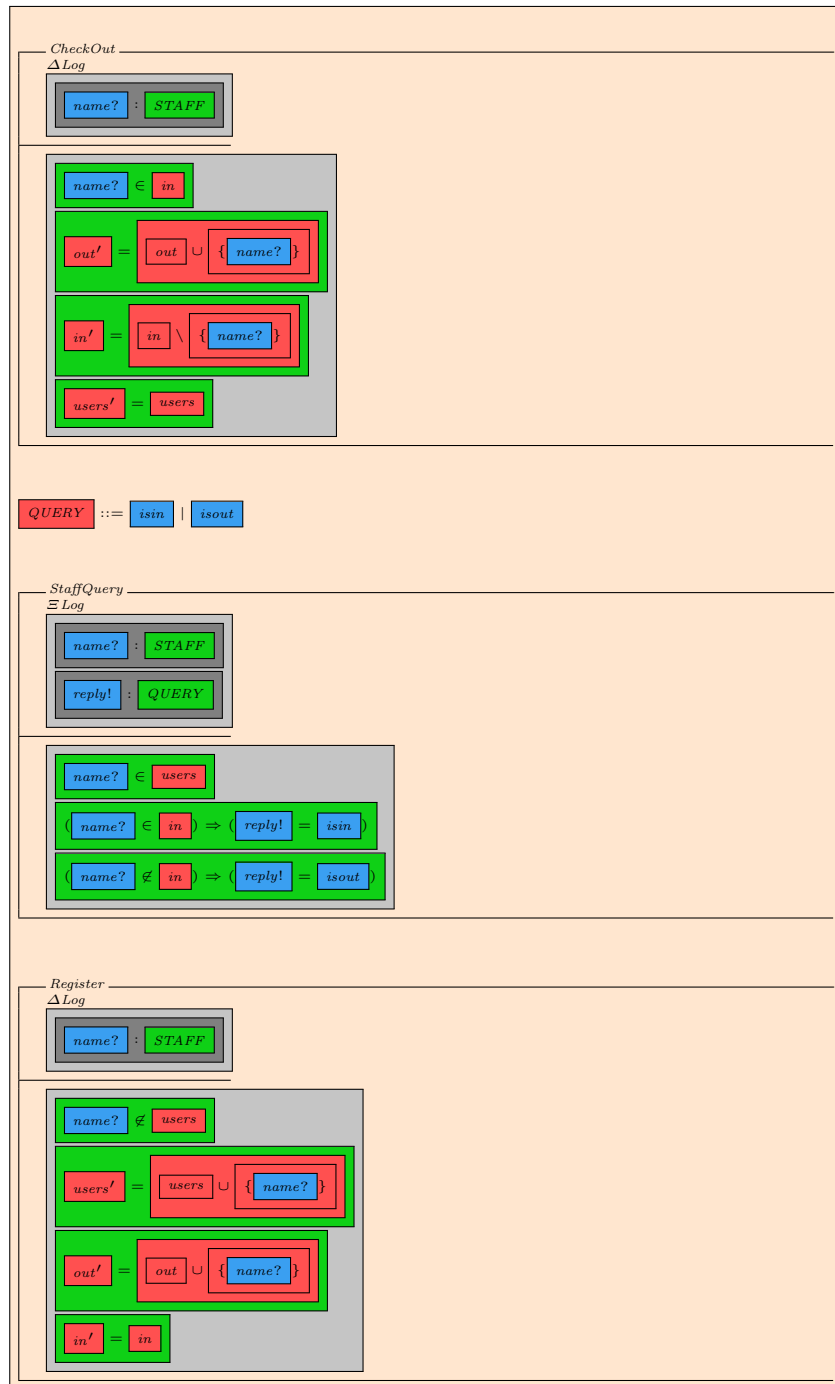
4. Dijkstra, E. W., Notes on Structured Programming. Technological University Eindhoven, Department of Mathematics, 1970.
5. Frentiu, M., Correctness, a very important quality factor in programming. *Studia Universitas Babes-Bolyai, Seria Informatica* L(1), 11–20, 2005.
6. ISO/IEC 13568, Information technology Z formal specification notation Syntax, type system and semantics, 2002,
7. Kamareddine, F., Wells, J., Zengler, C. and Barendregt, H.: Computerising Mathematical Text. *Computational Logic. Handbook of the History of Logic* 9, 343-396, Elsevier, 2014.
8. Kimber, T., Object-Z to Perfect Developer. Masters Thesis, Imperial College London, 2007.
9. Lamar, R., A Partial Translation Path from MathLang to Isabelle. PHD thesis, Heriot Watt University, 2011.
10. Li, Y., Pan, X., Hu, T., Sung, S. Y., Yuan, H., Specifying Complex Systems in Object-Z: A Case Study of Petrol Supply Systems. *Journal of Software*, 9(7), 2014.
11. MacKenzie, D., Computer-related accidental death: an empirical exploration, *Science and Public Policy*, 21(5), 233–248, 1994.
12. De Millo, Richard A. and Lipton, Richard J. and Perlis, Alan J, Social Processes and Proofs of Theorems and Programs, *Commun. ACM*, 22(5), 271–280, 1979.
13. Smith, G., The Object-Z Specification Language. Software Verification Research Centre, University of Queensland, 1999.
14. Parker, T.: TOZE - A Graphical Editor for the Object-Z Specification Language with Syntax and Type Checking Capabilities. Masters Thesis, University of Wisconsin-La Crosse, 2008.
15. http://www.eschertech.com/products/perfect_developer.php (Accessed April 2015).
16. Preibusch, S., Kamller, F.: Checking the TWIN Elevator System by Translating Object-Z to SMV. Formal Methods for Industrial Critical Systems: 12th International Workshop, FMICS, pp 38-57, 2007.
17. Jones, R., Methods and Tools for the Verification of Critical Properties, In 5th refinement workshop, springer workshops in computing, 2004.
18. Arthan, R., On Formal Specification of a Proof Tool. Lemma 1 Ltd.
19. oz Style File, <http://web.mit.edu/tex/stuff/latex-dist/psnfss/oz.sty> (Accessed August 2015).
20. Spivey, M., *notation: a reference manual*, Prentice-Hall, 1989.
21. Community Z Tools, <http://czt.sourceforge.net/> (Accessed August 2015).

A Credit Card ZCGa

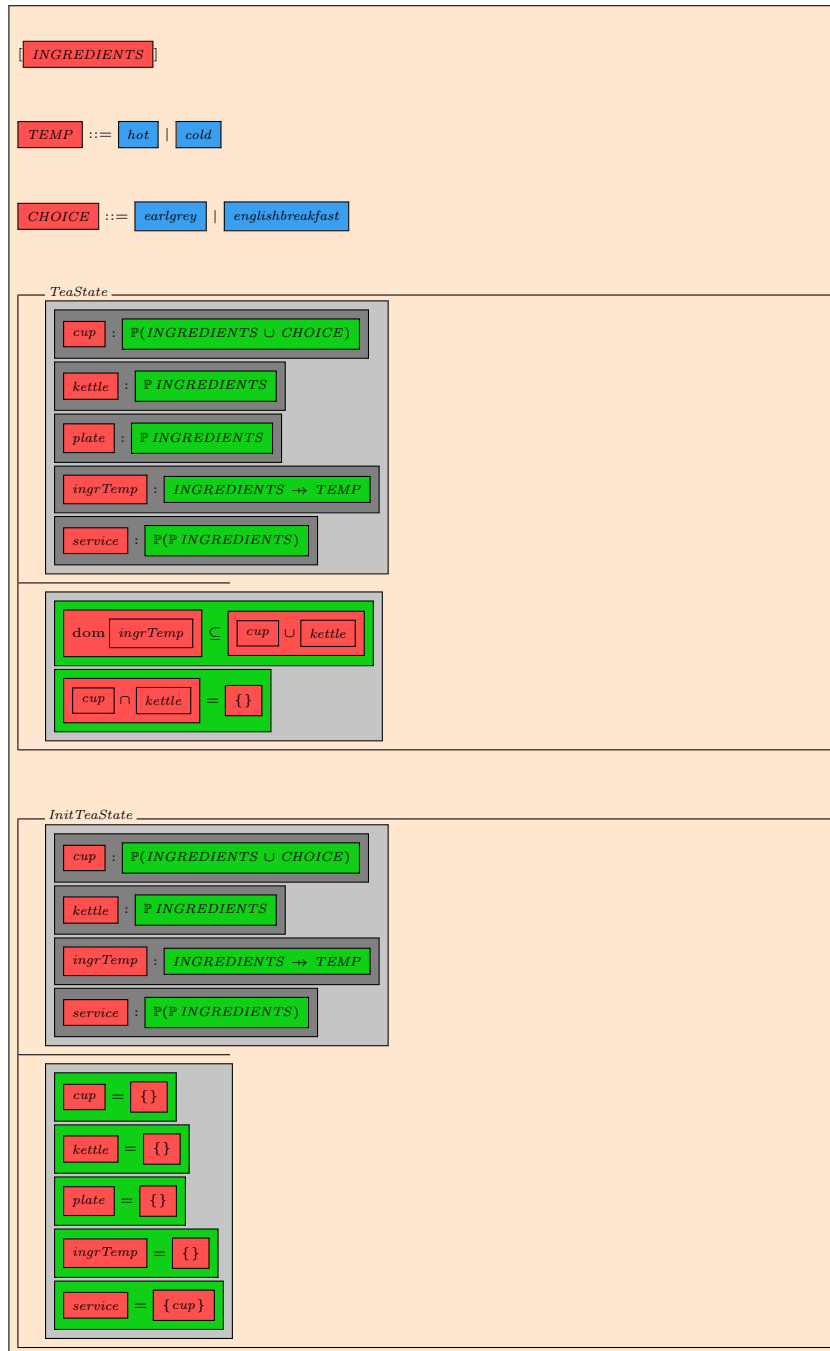


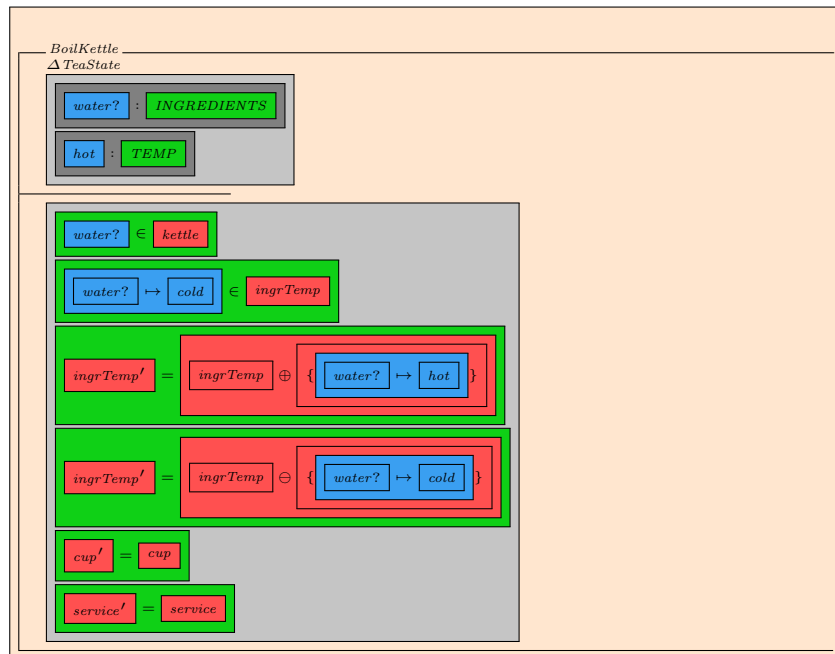
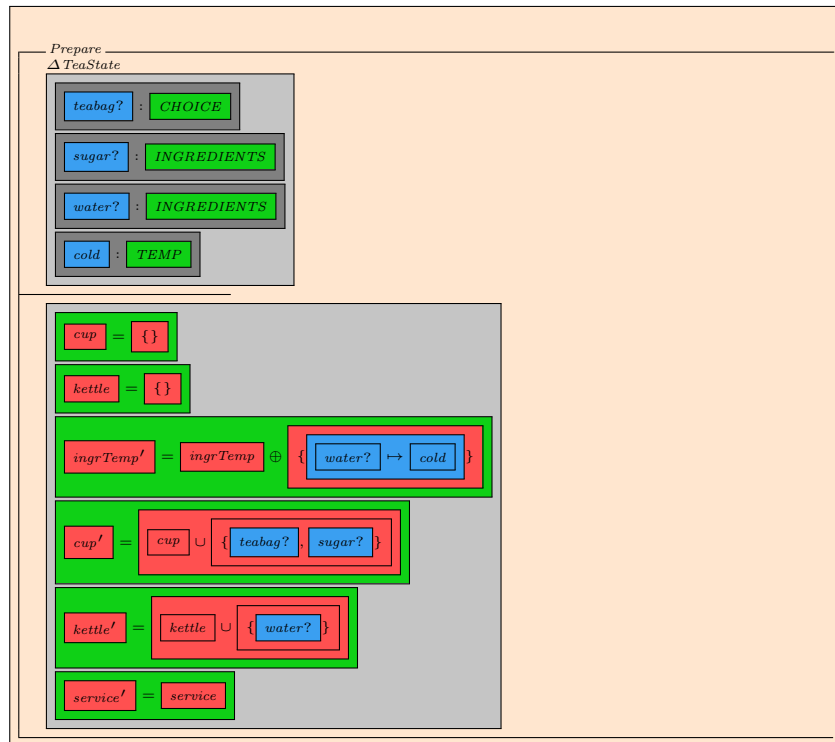
B CheckIn ZCGa

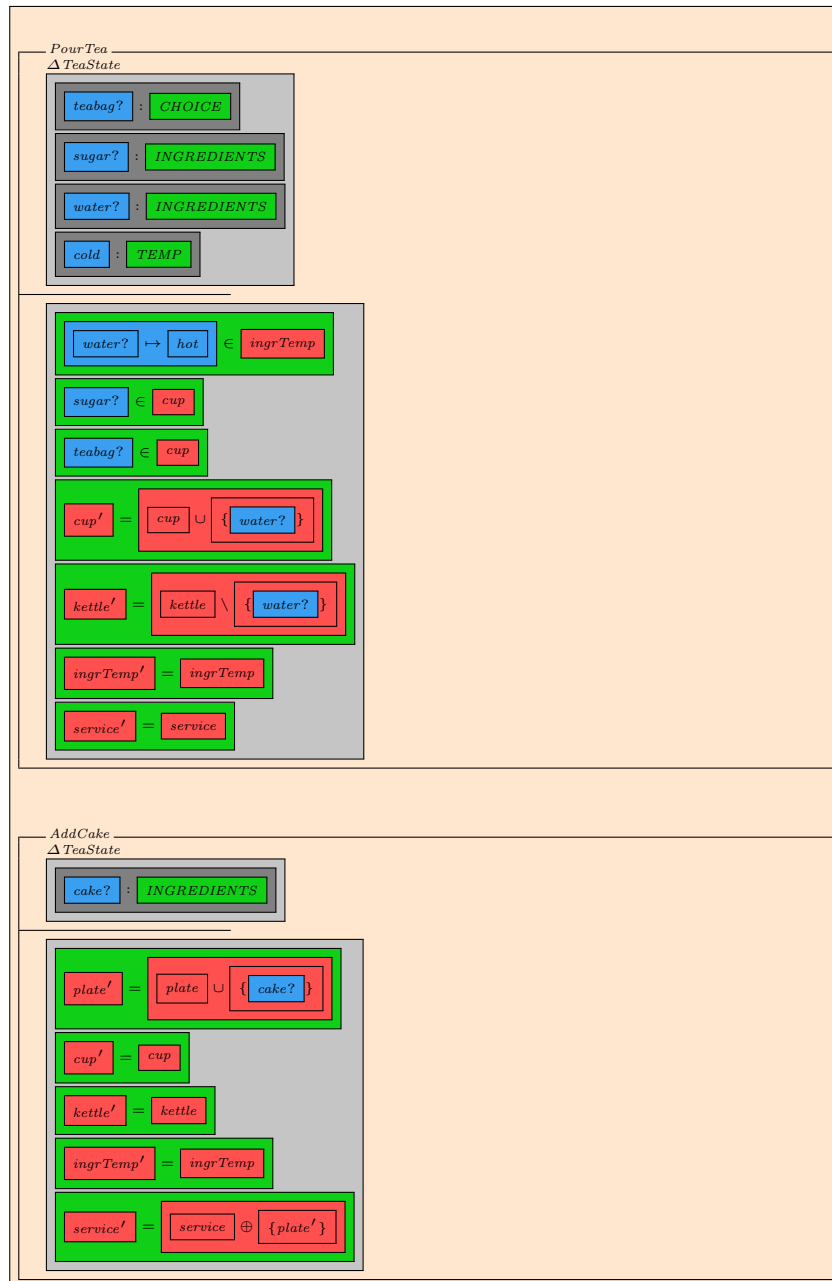


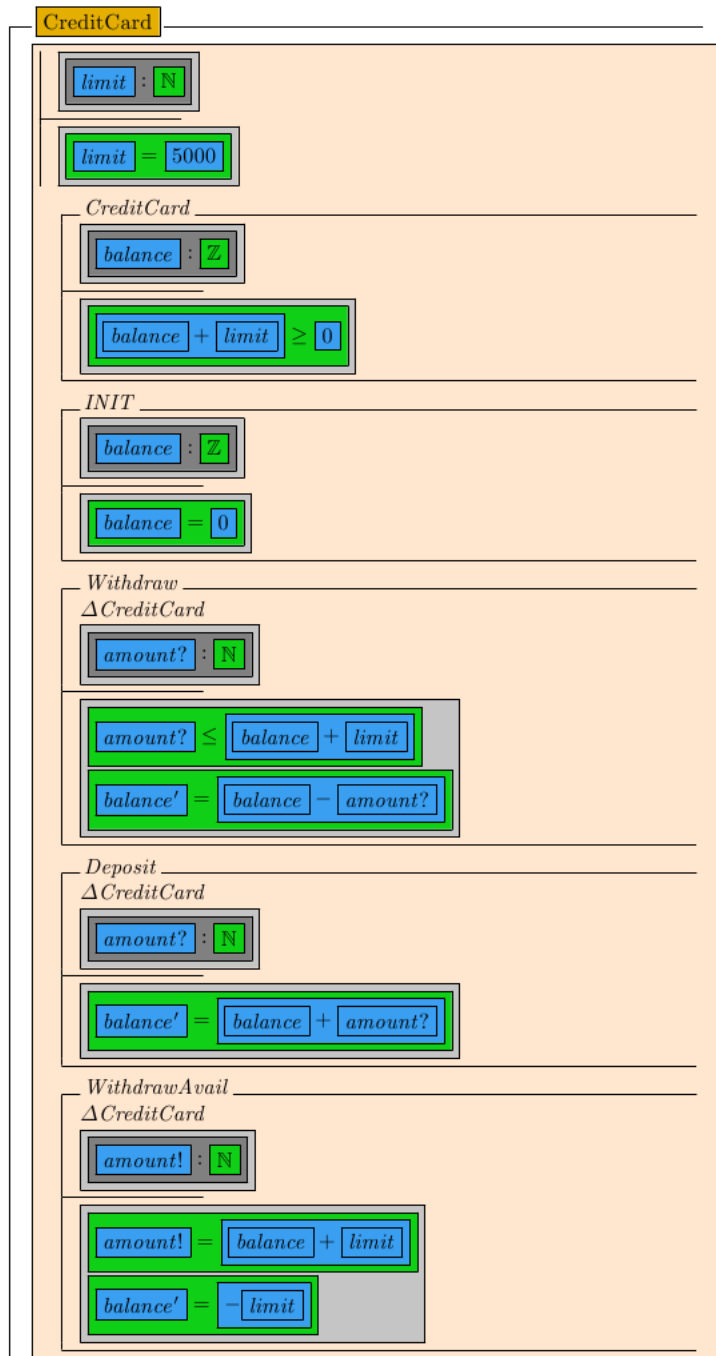


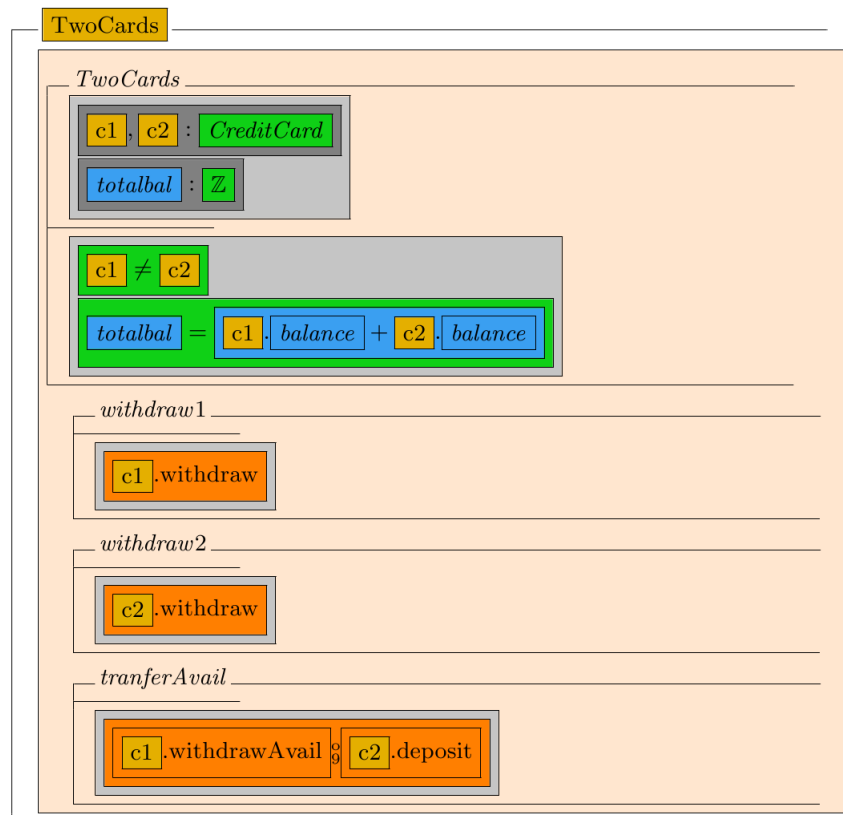
C Rich Tea ZCGa







D OZCGa L^AT_EX file



E A Breakdown of the Specification for Checking

