Generating natural language descriptions of Z test cases

Conference or Workshop Item

Generating Natural Language Descriptions of Z Test Cases

Maximiliano Cristiá
Flowgate Consulting and CIFASIS
Rosario, Argentina
mcristia@flowgate.net

Brian Plüss
Centre for Research in Computing
The Open University
Milton Keynes, UK
b.pluss@open.ac.uk

Abstract

Critical software most often requires an independent validation and verification (IVV). IVV is usually performed by domain experts, who are not familiar with specific, many times formal, development technologies. In addition, model-based testing (MBT) is a promising testing technique for the verification of critical software. Test cases generated by MBT tools are logical descriptions. The problem is, then, to provide natural language (NL) descriptions of these test cases, making them accessible to domain experts. In this paper, we present ongoing research aimed at finding a suitable method for generating NL descriptions from test cases in a formal specification language. A first prototype has been developed and applied to a real-world project in the aerospace sector.

1 Introduction

Model-based testing (MBT) is an active research area and a promising theory of software and hardware testing (Utting and Legeard, 2006; Hierons et al., 2009). MBT approaches start with a formal model or specification of the software, from which test cases are generated. These techniques have been developed and applied to models written in different formal notations, such as Z (Stocks and Carrington, 1996), finite state machines and their extensions (Grieskamp et al., 2002), B (Legeard et al., 2002), algebraic specifications (Bernot et al., 1991), and so on.

The fundamental hypothesis behind MBT is that, as a program is correct if it verifies its specification, then the specification is an excellent source of test cases. Once test cases are derived from the model, they are refined to the level of the implementation language and executed. The resulting output is then abstracted to the level of the specification language, and the model is used again to verify if the test case has detected an error.

The Test Template Framework (TTF) described by Stocks and Carrington (1996) is a particular MBT theory specially well suited for unit testing. The TTF uses Z specifications (Spivey, 1989) as the entry models and prescribes how to generate test cases for each operation included in the model. Fastest (Cristiá and Rodríguez Monetti, 2009) implements the TTF allowing users to automatically produce test cases for a given Z specification. Recently, we used Fastest to test an on-board satellite software for a major aerospace company in South America. Since Fastest uses models written in the Z specification language, test cases generated by this tool are paragraphs of formal text (see Section 2). This description is suitable for the automatic tasks involved in testing (e.g., automatic execution, hyperlinking, traceability), but humans need to be able to read Z specifications in order to understand what is being tested. In projects where independent verification and validation (IVV) is required this might be a problem, as most stakeholders will not necessarily be fluent in Z.

This is precisely the case in the project mentioned above, where the aerospace company requested not only the test cases in Z, but also in English. As it can be expected, in a project with hundreds of test cases, manual translation would increase the overall cost of testing and, most critically, reduce its quality due to the introduction of human errors. Interestingly, this problem is opposite to those in mainstream industrial practice, where test cases are described in natural language and must be formalised, in order to augment the quality and, hopefully, reduce the costs of testing.

Given the formal, structured nature of the source text, natural language generation (NLG) techniques seem to be an appropriate approach to solving this problem. In the rest of the paper, we give an example of a test case from the project mentioned above (Section 2), describe a template-based method for generating NL descriptions (Section 3), and propose further work towards a more general NLG solution (Section 4).

2 An Example from the Aerospace Industry

The problem of generating NL descriptions of specifications in Z arises in the following scenario: a company developing the software for a satellite needs to verify that the implementation conforms to a certain aerospace standard (ECSS, 2003) describing the basic functionality of any satellite software. We therefore started by modelling in Z the services described by the standard and used the Fastest tool to generate test cases.

The model is a "standard" Z specification: it has a *schema box* that defines the state space of the system and *operations* defining the transition relation between states[1]. Each operation formalizes one of the services described by the standard (e.g., memory dump, telecommand verification, enabling or disabling on-board monitoring, etc.).

Figure 1 shows one of the test cases generated for the operation *DumpMemoryAbsAdd*, that models a remote request for the on-board software to dump some portion of its memory. In TTF and Fastest, a Z test case is essentially a set of bindings between variables and values, and test cases are grouped according to the operation they test. Identifiers appearing in a test case are the input and state variables from the definition of the operation. These are bound to certain values defining the state in which the system must be tested and the input given to the operation in each unique test case. In the example, input variables are those *decorated* with a question mark, while state variables are plain identifiers. All these variables are declared somewhere else in the specification, by using a special schema box called *valid input space*, associated with each operation.

For example, the Z schema in Figure 1 indicates that the implementation of the dump memory service must be tested in a state where the system is processing a telecommand ($processingTC = yes$), the telecommand is a request for a memory dump ($srv = DMMA$), the system has one memory block

$$\begin{array}{l}
\textit{DumpMemoryAbsAdd\_SP\_7\_TCASE} \\
\hline
mid = mid0 \land srv = DMAA \land lengths = \varnothing \\
processingTC = yes \land adds = \varnothing \\
blocks = \{mid0 \mapsto \{1 \mapsto byte0, 2 \mapsto byte1, \\
\qquad\qquad 3 \mapsto byte2, 4 \mapsto byte3\}\} \\
m? = mid0 \land sa? = \langle 1 \rangle \land len? = \langle 2 \rangle \\
\hline
\end{array}$$

Figure 1: A test case described in Z

which is four bytes long ($blocks = \{\ldots\}$), there are no other pending requests ($adds = lengths = \varnothing$); and the request is for a memory dump of length two ($len? = \langle 2 \rangle$) starting at the first address ($sa? = \langle 1 \rangle$) of the available memory block ($m? = mid0 = mid$).

Fastest generated almost 200 test cases like the one depicted in Figure 1 from a model describing a simplified version of five services listed in the standard. The customer requested to deliver a natural language description of each one of them and a model describing all the services would have thousands of test cases. Clearly, trying to make the translation by hand would have been not only a source of errors, but also a technical retreat.

3 A Template-Based NLG Solution

As a first approach, we used a template-based method. We started by defining a grammar to express what we called *NL test case templates* (NLTCT). It appears in Figure 2[2]. Each NLTCT specifies how an NL description is generated for the Z test cases of a given operation. It starts with the name of the operation. Next follows a text section, intended as a parametrized NL description of the test case, where calls to *translation rules* can be inserted as needed. Finally, all necessary translation rules are defined, by indicating what is written in replacement for a call when a certain variable in the formal description of a test case appears bound to a specific value. In this way, a different text is generated for each test case, according to the binding between values and variables that defines the case. The Appendix shows the NLTCT for the operation *DumpMemoryAbsAdd*.

We implemented a parser in `awk` that takes an NLTCT and a Z test case, and generates the NL description of each test case in the input. Figure 3 shows the result for the test case in Figure 1.

This first prototype showed that NLTCTs tend

[1]The Z specification language is essentially typed first order logic, with syntactic sugar in the form of operators, that serve as shortcuts for frequently used complex expressions.

[2]Fastest saves formal test cases in text files written in ZLaTeX, an extension of the LaTeX markup language, what explains the use of this format in the NLTCT grammar.

```
      NLTCT ::= ⟨Operation⟩ eol
                ⟨NLTCD⟩ eol
                ⟨TCRule⟩{, ⟨TCRule⟩}
  Operation ::= operation =⟨identifier⟩
      NLTCD ::= \begin{tcase} eol
                ⟨LaTeXText⟩ eol
                \end{tcase}
  LaTeXText ::= LaTeX | ⟨TCRuleCall⟩ | ⟨LaTeXText⟩
 TCRuleCall ::= &rule ⟨identifier⟩ &
     TCRule ::= \begin{trule}{⟨identifier⟩} eol
                case ⟨identifier⟩[, ⟨identifier⟩] eol
                ⟨RuleDef⟩ eol {, ⟨RuleDef⟩ eol}
                endcase eol
                \end{trule}
    RuleDef ::= $⟨ZLaTeX⟩[" | " ⟨ZLaTeX⟩ | & ⟨ZLaTeX⟩]
                : ⟨LaTeX⟩ eol
      LaTeX ::= free LaTeX text
     ZLaTeX ::= free Z LaTeX text
```

Figure 2: Grammar for NLTC templates

to be relatively small and simple, in spite of the large number of test cases. This is due to test cases combining a small set of values in many different ways. However, NLTCTs for large operations tend to become increasingly more complex, for the number of combinations grows exponentially. As a consequence, these operations require a large number of cases within translation rules and sometimes even more translation rules[3].

A thorough evaluation of this method is due. Its suitability must be measured from the perspective of two kinds of users: (a) the engineers who write the formal models, generate the formal test cases and write the NLTCTs; and (b) other stakeholders (e.g., the customer, auditors, domain experts), who read the descriptions of the test cases in natural language. For the engineers, applying the method should be more efficient, in terms of time and effort, than writing the descriptions by hand. For the readers, success will be determined by the readability of the output and, more critically, by its precision with respect to the specification. At the moment of writing, we are designing two empirical studies aimed at obtaining these measures.

4 Future and Related Work

The solution presented above was successful in generating adequate NL descriptions of the test cases in one particular project. However, the limitations mentioned in the previous section show that this solution would not generalise well to specifications in other domains. Moreover, it requires defining a new template for each operation; a task of still considerable size for large systems.

[3]This is because templates are written in terms of the values bound to variables, and not in terms of the predicates satisfied by those values, which are nonetheless available as part of the MBT approach.

Test case: DumpMemoryAbsAdd_SP_7_TCASE

Service (6,5) will be tested in a situation that verifies that:

- the state is such that:
  - the on-board system is currently processing a telecommand and has not answered it yet.
  - the service type of the telecommand is DMAA.
  - the set of sequences of available memory cells contains only one sequence, associated to a memory ID, which has four different bytes.
  - the set of starting addresses of the chunks of memory that have been requested by the ground is empty.
- the input memory ID to be dumped is the available memory ID, the input set of start addresses of the memory regions to be dumped is the unitary sequence composed of 1, the set of numbers of memory cells to be dumped is the unitary sequence composed of 2.

Figure 3: NL description of the test in Figure 1

At the same time, Z specifications contain all the information necessary to produce the templates for the operations in the system, regardless of its domain of application. This information is structured according to the syntax of the formal language. Additionally, when formally specifying a system, it is common practice to include associations between the identifiers in the specification (new types, operations, state schemata, variables, constants, etc.) and the elements they refer to in the application domain (i.e., aerospace software). These associations are called *designations* (Jackson, 1995), some of which, relevant to the test case in Figure 1, are shown in Figure 4.

These considerations lead us to believe in the

srv	$\approx$	Service type of the telecommand
DMAA	$\approx$	Dump memory using absolute addresses
processingTC	$\approx$	The on-board system is currently processing a telecommand and has not answered it yet
m?	$\approx$	Memory ID to be dumped
sa?	$\approx$	Start addresses of the memory regions to be dumped
len?	$\approx$	The number of memory cells to be dumped for each start address

Figure 4: Designations for the test in Figure 1

possibility of generating NL descriptions of Z test cases automatically by using their definitions, the system specification and the designations of identifiers. Such a solution would be independent of the application domain and, more importantly, of the number of operations in the model.

The linguistic properties of the target document are relevant in devising an adequate treatment for the input, but the overall structure of the output remains rigid and its content is determined by the definition of each test case. The approach would still be template-based, but in terms of the NLG architecture of Reiter and Dale (2000), templates would be defined at the level of the document structure[4], with minimal microplanning and surface strings generated according to the part of the test case being processed and the designations of the identifiers[5]. The next stages of our project will point in this direction, using techniques from NLG for automating the definition of the templates presented in the previous section.

There have been efforts for producing natural language versions of formal specifications in the past. Punshon et al. (1997) use a case study to present the REVIEW system (Salek et al., 1994)[6]. REVIEW automatically paraphrases specifications developed with Metaview (Sorenson et al., 1988), an academic research *metasystem* that facilitates the construction of CASE environments to support software specification tasks. Coscoy (1997) describes a mechanism based on program extraction, for generating explanations of formal proofs in the Calculus of Inductive Constructions, implemented in the Coq Proof Assistant (Bertot and Castéran, 2004). Lavoie et al. (1997) present MODEX, a tool that generates customizable descriptions of the relations between classes in object-oriented models specified in the ODL standard (Cattell and Barry, 1997). Bertani et al. (1999) describe a controlled natural language approach to translating formal specifications written in an extension of TRIO (Ghezzi et al., 1990) by transforming syntactic trees in TRIO into syntactic trees of the controlled language.

The solutions presented in the related work above are highly dependant on particular aspects of the source language and do not apply directly to specifications written in Z. To our knowledge, no work has been done towards producing NL descriptions of Z specifications. The same holds for test cases generated using the MBT approach.

[4]Somewhat along the lines of what Wilcock (2005) describes for XML-based NLG.

[5]This approach is similar to the method proposed by Kittredge and Lavoie (1998) for generating weather forecasts.

[6]Salek et al. (1994) also give a comprehensive survey of related work for generating NL explanations for particular specification languages (most of which are now obsolete).

5 Conclusion

In this paper we presented a concrete NLG problem in the area of software development involving formal methods. We focused the description on the generation of NL descriptions of test cases, but nothing prevents us from extending the idea to entire system specifications.

The development of a general technique for verbalising formal specification would fill the communication gap between system designers and other stakeholders in the development process, while preserving the advantages associated to the use of formal methods: precision, lack of ambiguity, formal proof of system properties, etc.

Finally, we hope this paper draws attention from NLG experts to an area which would benefit substantially from their expertise.

Acnowledgements

A substantial part of this research was funded by Flowgate Consulting. We would also like to thank Richard Power from the NLG group at The Open University for help in finding financial support, Eva Banik for helpful comments on earlier versions of this paper, and three anonymous reviewers for useful feedback and suggestions.

References

G. Bernot, M.C. Gaudel, and B. Marre. 1991. Software testing based on formal specifications: a theory and a tool. *Software Engineering Journal (SEJ)*, 6(6):387–405.

A. Bertani, W. Castelnovo, E. Ciapessoni, and G. Mauri. 1999. Natural language translations of formal specifications for complex industrial systems. In *AI*IA 1992: Proceedings of the 6th Congress of the Italian Association for Artificial Intelligence*, pages 185–194, Bologna, Italy.

Y. Bertot and P. Castéran. 2004. *Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions*. Texts in Theoretical Computer Science. Springer-Verlag.

R.G.G. Cattell and D.K. Barry, editors. 1997. *The object database standard: ODMG 2.0*. Morgan Kaufmann Publishers Inc., San Francisco, CA.

Y. Coscoy. 1997. A natural language explanation for formal proofs. In *LACL '96: Selected papers from the First International Conference on Logical Aspects of Computational Linguistics*, pages 149–167, London, UK. Springer-Verlag.

M. Cristiá and P. Rodríguez Monetti. 2009. Implementing and applying the Stocks-Carrington framework for model-based testing. In Karin Breitman and Ana Cavalcanti, editors, *ICFEM*, volume 5885 of *Lecture Notes in Computer Science*, pages 167–185. Springer-Verlag.

ECSS. 2003. Space Engineering – Ground Systems and Operations: Telemetry and Telecommand Packet Utilization. Technical Report ECSS-E-70-41A, European Space Agency.

C. Ghezzi, D. Mandrioli, and A. Morzenti. 1990. TRIO: A logic language for executable specifications of real-time systems. *Journal of Systems and Software*, 12(2):107–123.

W. Grieskamp, Y. Gurevich, W. Schulte, and M. Veanes. 2002. Generating finite state machines from abstract state machines. In *ISSTA '02: Proceedings of the 2002 ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 112–122, Rome, Italy.

R.M. Hierons, K. Bogdanov, J.P. Bowen, R. Cleaveland, J. Derrick, et al. 2009. Using formal specifications to support testing. *ACM Computing Surveys (CSUR)*, 41(2):9.

M. Jackson. 1995. *Software requirements & specifications: a lexicon of practice, principles, and prejudices*. Addison-Wesley.

R. Kittredge and B. Lavoie. 1998. Meteocogent: A knowledge-based tool for generating weather forecast texts. In *Proceedings of American Meteorological Society AI Conference (AMS-98)*, Phoenix, AZ.

B. Lavoie, O. Rambow, and E. Reiter. 1997. Customizable descriptions of object-oriented models. In *Proceedings of the Conference on Applied Natural Language Processing (ANLP'97*, pages 253–256, Washington, DC.

B. Legeard, F. Peureux, and M. Utting. 2002. A Comparison of the BTT and TTF Test-Generation Methods. In *ZB '02: Proceedings of the 2nd International Conference of B and Z Users on Formal Specification and Development in Z and B*, pages 309–329, London, UK. Springer-Verlag.

J.M. Punshon, J.P Tremblay, P.G. Sorenson, and P.S. Findeisen. 1997. From formal specifications to natural language: A case study. In *12th IEEE International Conference Automated Software Engineering*, pages 309–310.

E. Reiter and Robert Dale. 2000. *Building Natural Language Generation Systems*. Cambridge University Press, Cambridge, UK.

A. Salek, P.G. Sorenson, J.P. Tremblay, and J.M. Punshon. 1994. The REVIEW system: From formal specifications to natural language. In *Proceedings of the First International Conference on Requirements Engineering*, pages 220–229.

P.G. Sorenson, J.P. Tremblay, and A.J. McAllister. 1988. The Metaview system for many specification environments. *IEEE Software*, 5(2):30–38.

J.M. Spivey. 1989. *The Z Notation: A Reference Manual*. Prentice-Hall, Inc.

P. Stocks and D. Carrington. 1996. A Framework for Specification-Based Testing. *IEEE Transactions on Software Engineering*, 22(11):777–793.

M. Utting and B. Legeard. 2006. *Practical Model-Based Testing: A Tools Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA.

G. Wilcock. 2005. An Overview of Shallow XML-based Natural Language Generation. In *Proceedings of the 2nd Baltic Conference on Human Language Technolgies*, pages 67–78, Tallinn, Estonia.

Appendix A. NLTCT for the Example

NLTCT for *DumpMemoryAbsAdd* (some parts were replaced by `[...]` due to space restrictions):

```
operation = DumpMemoryAbsAdd

\begin{tcase}
\centerline{{\bf Test case: \ltcaseid}}

The service (6,5) will be tested in the
situation that verifies that:
\begin{itemize}
  \item the state is such that:
  \begin{itemize}
    \item the on-board system is &trule PTCr&.
    \item the service type of the telecommand
          is &trule SRVr&.
    [...]
    \item the set of starting addresses of the
          chunks of memory that have been
          requested by the ground is &trule
          ADSr&.
  \end{itemize}
  [...]
\end{itemize}
\end{tcase}

\begin{trule}{PTCr}
case processingTC
$yes :currently processing a telecommand and
      has not answered it yet
$no :not currently processing a telecommand
endcase
\end{trule}

\begin{trule}{SRVr}
case srv
$* :*
endcase
\end{trule}

\begin{trule}{ADSr}
case adds
$\emptyset :empty
$\langle 0 \rangle :the unitary sequence
                    composed of 0
endcase
\end{trule}

[...]
```