Applying the Test Template Framework to Aerospace Software

Maximiliano Cristiá[*†‡], Pablo Albertengo [*], Claudia Frydman[‡], Brian Plüss[§] and Pablo Rodríguez Monetti[*¶]
[*]Flowgate Consulting, Rosario, Argentina
[†]CIFASIS-UNR, Rosario Argentina
[‡]LSIS-CIFASIS, Marseille, France
[§]The Open University, Milton Keynes, UK
[§]FCEIA-UNR, Rosario, Argentina

***Abstract*—We have applied Fastest, an implementation of the Test Template Framework, to five real case studies of aerospace software. This involved the formalization in the Z notation of nontrivial parts of each system. One of these models, for instance, formalizes a significant portion of the ECSS-E-70-41A aerospace standard. The models were then fed into Fastest, which automatically generated detailed functional abstract test cases. Since these test cases are independent of any implementation, they can be used to test any of them. Furthermore, we were able to semi-automatically translate them into English so they can be used by domain experts performing independent validation and verification activities.**

I. Model-Based Testing of Aerospace Software

This paper reports on the application of a particular model-based testing (MBT) method, and a tool implementing it, to five real case studies from the aerospace domain. MBT is a well-known approach aimed at testing software systems starting with a formal model [1], [2], from which test cases are generated. The MBT method we applied, known as the Test Template Framework (TTF) [3]–[5], works with Z specifications and is specially well-suited for unit testing. Fastest [6]–[8] is the first tool implementing the TTF.

Applying Fastest to the case studies involved writing formal models of parts of these problems and then using the tool to generate test specifications and abstract test cases. Although, the MBT process does not end in producing abstract test cases, so far the MBT community has been focused on methods and techniques to generate test cases at the level of the model [2]. The intention of this report is, then, to contribute to the MBT community with five industrial-strength case studies regarding the generation of abstract test cases in a mission-critical application domain. Therefore, test case refinement, test case execution and the remaining steps of the MBT process are beyond the scope of this article. We want to emphasize that this paper is a summary of several technical documents which amounts to a total of 440 pages, making it impossible to show here more than the most important aspects of them. Our intention is that readers take this paper as a summary pointing to the technical documentation available at http://www.flowgate.net/pdf/reports.tar.gz.

We want to remark that the TTF and Fastest generate unit tests, which is not the most common approach in the MBT community [2]. Therefore, the results shown in this paper should be of interest for those readers looking for formal approaches to unit testing.

Fastest is a command-line application that reads Z specifications written in Standard Z LaTeX (SZL) and returns abstract test cases in SZL generated according to the TTF. From the command-line users can issue commands to follow the steps of the TTF. Z specifications can be written with any text editor, however we suggest to use Eclipse [9] with the Community Z Tools (CZT) and TeXlipse plugins [10], [11], because they allow for easy checking of the LaTeX syntax and Z syntax and semantics, as well. Nevertheless, Fastest checks the syntax and semantics while specifications are loaded, and prompt the errors they might have. Once the specification is successfully loaded, users can issue a number of commands to produce abstract test cases. The user's guide delivered with the tool [12] includes a detailed description of each command. Fastest is open-source software available at [13].

Section II introduces the TTF by means of an example from the aerospace domain. Section III introduces a case study regarding an on-board flight control system; Section IV show two case studies about satellite on-board communication protocols; and Section V contains the results of applying our approach to a relevant portion of the ECSS-E-70-41A standard in two different settings. Our conclusions are in Section VI.

II. The Test Template Framework and Fastest

In this section we briefly introduce the TTF and Fastest by means of an example from the aerospace domain—for further details see [3]–[8]. We assume the reader has a basic knowledge of the Z formal notation—otherwise, consult [14], [15]—, and of the ECSS-E-70-41A Standard [16]. In particular, we will use the terminology defined in [16] without further introduction.

Consider the "Enable Storage in Packet Stores (15,1)" service subtype [16, pages 156-157]. If an on-board satellite software is to implement this functionality, then it must be able to receive a telecommand with two application data fields:

– N: is 'the number of packet stores to be controlled' (unsigned integer) [16].
– Store ID: is a fixed char string repeated N times indicating 'for example, an access path to a physical on-board recording device or file' [16].

$$[StID, StAttr]$$
$$FAILCODE ::= OK \mid TPE$$

$$\begin{array}{|l}
\textit{Stores} \\ \hline
eSt : \mathbb{P}\, StID \\
stDef : StID \rightarrowtail\!\!\!\!\!+ \; StAttr \\ \hline
\end{array}$$

$$\begin{array}{|l}
\textit{EnableStE} \\ \hline
\Xi Stores \\
ps? : \mathbb{P}\, StID \\
code! : FAILCODE \\ \hline
\neg\,(ps? \subseteq \operatorname{dom} stDef \\
\qquad \wedge\, eSt \cap ps? = \varnothing) \\
code! = TPE \\ \hline
\end{array}$$

$$\begin{array}{|l}
\textit{EnableStOk} \\ \hline
\Delta Stores \\
ps? : \mathbb{P}\, StID;\ code! : FAILCODE \\ \hline
ps? \subseteq \operatorname{dom} stDef \\
eSt \cap ps? = \varnothing \\
eSt' = eSt \cup ps? \\
code! = OK \\ \hline
\end{array}$$

$$EnableSt == EnableStOk \vee EnableStE$$

Fig. 1. A Z specification for the "Enable Storage in Packet Stores" sub-service [16].

When this telecommand is received the software 'shall start sending the relevant packets to the application processes managing the specified packet stores' [16].

Figure 1 shows a simplified version of a Z specification of these requirements. In this model, schema *Stores* represents the state space. State variable *eSt* is the set of stores ID's that have been enabled so far; and *stDef* returns the store attributes for a given store ID—enabled or not. A more expressive definition of *StAttr* is omitted for brevity. Schema *EnableSt* is the specification of the requirements stated above. The operation is divided into two schemas: *EnableStOk*, which adds the store IDs received in *ps*? to the set of enabled stores, which in turn enables other operations to use these packet stores; and *EnableStE*, which specifies that an error message must be returned if the preconditions are not met. As it can be seen, both application data fields are combined into one input variable, *ps*?, whose type is $\mathbb{P}\, StID$ since it receives the store ID's and $\#ps?$ equals N. Furthermore, the requirement asking Store ID to be char strings is abstracted as a given Z type, *StID*, whose structure is unknown—as in fact suggested by the standard: 'the meaning and internal structure of the Store ID are beyond the scope of this Standard'.

The idea behind the TTF, as well as of other MBT methods, is to analyze a Z specification deriving abstract test cases—i.e., test cases written in Z—that are later used to test the corresponding implementation. Within the TTF, the *valid input space* (VIS) of a Z operation is partitioned into so-called *test classes*[1] by means of *testing tactics* to form a *testing tree*; from the leaves of such a tree, abstract test cases must be derived. The VIS of an operation is the Z schema constituted by all its input and state variables. In the example we have:

$$\begin{array}{l}
EnableSt_{VIS} == \\
\qquad [eSt : \mathbb{P}\, StID;\ stDef : StID \rightarrowtail\!\!\!\!\!+ \; StAttr;\ ps? : \mathbb{P}\, StID]
\end{array}$$

A testing tactic is applied to partition the VIS, giving rise to a set of test classes. A second testing tactic can be applied to one or more of these test classes, giving rise to a second set. This process can continue until the tester is satisfied. All these test classes are organized in a testing tree. There are a number of tactics proposed by the TTF and available in Fastest. Since Disjunctive Normal Form (DNF) warrants a minimum coverage, Fastest applies it as the first tactic regardless of the user's choice. In the example, after applying DNF we have the following three test classes:

$$\begin{array}{l}
EnableSt_1^{DNF} == \\
\qquad [EnableSt_{VIS} \mid ps? \subseteq \operatorname{dom} stDef \wedge eSt \cap ps? = \varnothing] \\
EnableSt_2^{DNF} == [EnableSt_{VIS} \mid \neg\, ps? \subseteq \operatorname{dom} stDef] \\
EnableSt_3^{DNF} == [EnableSt_{VIS} \mid \neg\, eSt \cap ps? = \varnothing]
\end{array}$$

As it can be seen, each test class is a Z schema specifying a test condition. We then partition $EnableSt_1^{DNF}$ by applying the tactic Standard Partitions (SP) to the $\cup$ operator used in $eSt \cup ps?$. SP defines a standard partition for each mathematical operator by giving a set of conditions on its operands. The standard partition for the $\cup$ operator is shown in Figure 2. Therefore, we have the following new test classes:

$$\begin{array}{l}
EnableSt_1^{SP} == [EnableSt_1^{DNF} \mid eSt = \varnothing \wedge ps? = \varnothing] \\
EnableSt_2^{SP} == [EnableSt_1^{DNF} \mid eSt = \varnothing \wedge ps? \neq \varnothing] \\
EnableSt_3^{SP} == [EnableSt_1^{DNF} \mid eSt \neq \varnothing \wedge ps? = \varnothing] \\
EnableSt_4^{SP} == [EnableSt_1^{DNF} \mid \\
\qquad eSt \neq \varnothing \wedge ps? \neq \varnothing \wedge eSt \cap ps? = \varnothing] \\
EnableSt_5^{SP} == \\
\qquad [EnableSt_1^{DNF} \mid eSt \neq \varnothing \wedge ps? \neq \varnothing \wedge eSt \subset ps?] \\
EnableSt_6^{SP} == \\
\qquad [EnableSt_1^{DNF} \mid eSt \neq \varnothing \wedge ps? \neq \varnothing \wedge ps? \subset eSt] \\
EnableSt_7^{SP} == \\
\qquad [EnableSt_1^{DNF} \mid eSt \neq \varnothing \wedge ps? \neq \varnothing \wedge ps? = eSt]
\end{array}$$

$$\begin{array}{ll}
S = \varnothing, T = \varnothing & S \neq \varnothing, T \neq \varnothing, S \subset T \\
S = \varnothing, T \neq \varnothing & S \neq \varnothing, T \neq \varnothing, T \subset S \\
S \neq \varnothing, T = \varnothing & S \neq \varnothing, T \neq \varnothing, T = S \\
S \neq \varnothing, T \neq \varnothing, & S \neq \varnothing, T \neq \varnothing, S \cap T \neq \varnothing, \\
\quad S \cap T = \varnothing & \quad \neg\,(S \subseteq T), \neg\,(T \subseteq S), S \neq T
\end{array}$$

Fig. 2. Standard partition for expression of the form $S \cup T$.

[1]Also called *test objectives*, *test specifications*, *test templates*, *test conditions*, etc.

$$\begin{array}{l} EnableSt_8^{SP} == [EnableSt_1^{DNF} \mid \\ \quad eSt \neq \varnothing \land ps? \neq \varnothing \land eSt \cap ps? \neq \varnothing \\ \quad \land \neg\ eSt \subseteq ps? \land \neg\ ps? \subseteq eSt \land ps? \neq eSt] \end{array}$$

Note that the conditions defined by the new testing tactic are conjoined to the conditions of the parent node by schema inclusion—$EnableSt_1^{DNF}$ in this case. These representations of test classes are called testing trees.

As far as the original presentation of the TTF concerns, the choice of testing tactics to apply to a particular operation is a manual activity carried on by a testing engineer based on an analysis of the coverage the resulting test classes will give and the kind of errors they might uncover. DNF guarantees a minimum logical coverage since all the main functional alternatives will be exercised. However, errors in the implementation of complex mathematical operators such us $\cup$ have a low probability to be revealed by DNF since it does not take them into account. In this case, SP is a much better candidate since it looks a the semantics of these operators disregarding the logical structure of the predicate. Below we will introduce other testing tactics implemented in Fastest, which constitute a good menu to generate test classes for most of the Z specifications. Although the choice of tactics was thought as a manual activity, some heuristics can be implemented in a way that they are applied according to the elements present in each operation—this has not been investigated in the context of the TTF.

Usually, some of the test classes generated by the TTF are unsatisfiable—for instance $EnableSt_5^{SP}$ in the running example. Hence, Fastest implements a powerful method for detecting them. The core of this method is a library of so called elimination theorems, each of which represents a family of mathematical contradictions [7]. After running the pruning command, only $EnableSt_1^{SP}$ to $EnableSt_4^{SP}$, $EnableSt_2^{DNF}$ and $EnableSt_3^{DNF}$ remain. According to the TTF, at least one test case must be derived from each surviving leaf. In this context a test case is a tuple of constants satisfying the predicate of a given test class. Fastest implements a satisfability algorithm covering a significant number of the mathematical theories included in the Z mathematical toolkit; the method uses the ZLive component of the CZT project [17]. In this example, Fastest produces automatically the following test cases:

$$\begin{array}{l} EnableSt_1^{TC} == [EnableSt_1^{SP} \mid \\ \quad eSt = \varnothing \\ \quad \land stDef = \varnothing \land ps? = \varnothing] \\ EnableSt_2^{TC} == [EnableSt_2^{SP} \mid \\ \quad eSt = \varnothing \\ \quad \land stDef = \{(stid0, stattr0)\} \land ps? = \{stid0\}] \\ EnableSt_3^{TC} == [EnableSt_3^{SP} \mid \\ \quad eSt = \{stid0\} \land stDef = \varnothing \land ps? = \varnothing] \\ EnableSt_4^{TC} == [EnableSt_4^{SP} \mid \\ \quad eSt = \{stid0\} \\ \quad \land stDef = \{(stid1, stattr0)\} \land ps? = \{stid1\}] \\ EnableSt_5^{TC} == [EnableSt_2^{DNF} \mid \\ \quad eSt = \varnothing \land stDef = \varnothing \land ps? = \{stid0\}] \\ EnableSt_6^{TC} == [EnableSt_3^{DNF} \mid \\ \quad eSt = \{stid0\} \land stDef = \varnothing \land ps? = \{stid0\}] \end{array}$$

After writing the Z specification in the LaTeX markup defined in [18], the entire process is executed in Fastest via the script depicted in Figure 3.

A. More Testing Tactics

As we have said, within the TTF exists a number of testing tactics. In the case studies presented in this paper we have used the following ones.

Free Types (FT) is a testing tactic useful for dealing with variables whose type is enumerated. If x is a variable of an enumerated type, T, FT generates test classes whose characteristic predicates are of the form $x = val$ for each $val \in T$. In this way, the tactic guarantees that the implementation will be exercised on all these values, which are usually part of conditional expressions.

Numeric Ranges (NR) is a testing tactic that waits for an arithmetic variable, var, and an ordered list of numbers, $n_1, \dots n_k$, and generates the following partition: $var < n_1$, $var = n_1$, $n_1 < var < n_2$, $\dots$, $var = n_i$, $n_i < var < n_{i+1}$, $var = n_{i+1}$, $\dots$, $var < n_k$, $var = n_k$ and $n_k < var$.

With Mandatory Test Set (MTS) the user can bind a set of constants, $\{v_1, \dots, v_n\}$ to an expression, $expr$, in such a way that, when the tactic is applied, it generates $n + 1$ test classes characterized by the following predicates: $expr = v_i$ for all i in $1 \mathrel{..} n$, and $expr \notin \{v_1, \dots, v_n\}$. This tactic allows users to test the implementation on specific values that they deem important.

In Set Extension (ISE) is a tactic that applies to operations including predicates of the form $expr \in \{expr_1, \dots, expr_n\}$. In this case, it generates n test classes such that $expr = expr_i$, for i in $1 \mathrel{..} n$, are their characteristic predicates.

B. Test Oracles in the TTF

Since the TTF has several differences with other MBT methods, we consider it necessary to explain how it deals with test oracles. A test oracle is a means by which it is possible to determine whether a test case has found an error in the implementation or not. In the TTF the specification acts as a test oracle. In effect, according to the TTF: (a) abstract test cases must be refined into executable test cases [19]; (b) these test cases are executed by the program under test; (c) the output produced by the program for a given test case is abstracted at the level of the specification (if this is possible); and (d) both the abstract test case and its corresponding abstract output are substituted in the schema of the appropriate Z operation. In this way the predicate of the operation becomes a constant predicate: if it reduces to *true*, then no error was found in the implementation; otherwise the implementation is faulty on this test case. If it is impossible to abstract the output of the program (for example, because it ends with "segmentation fault"), then an error was found. Hence, in the TTF test cases do not include their oracles. In

```
loadspec enableStores.tex                     loads the specification; type-checks it
selop EnableSt                                generates test only for selected schemas
genalltt                                      applies DNF; builds testing tree
addtactic EnableSt_DNF_1 SP \cup eSt \cup ps?
genalltt                                      applies SP; rebuilds testing tree
prunett                                       prunes the testing tree
genalltca                                     generates test cases for the leaves
```

Fig. 3. Script to produce abstract test cases.

fact, an abstract test case only defines the values for each input variable.

C. State Invariants in the TTF

Usually Z specifications include a state invariant in the state schema. As the reader can see, we have not included any state invariant in *Stores*. We prefer writing state invariants as follows:

$$StoresInv == [Stores \mid eSt \subseteq \operatorname{dom} stDef]$$

and then asking for a proof obligation for each operation:

Theorem *EnableStPI*
$$StoresInv \land EnableSt \Rightarrow StoresInv'$$

When the operation is analysed by the TTF, it will generate test cases that will test code implementing sufficient functionality to make the program verify the state invariant, because it implements its specification and the specification satisfies the invariant. More formally: if operation O satisfies invariant I—i.e. $I \land O \Rightarrow I'$—and we "prove" by testing that program P implements O—i.e. $P \Rightarrow O$—, then we can prove that P satisfies I—i.e. $I \land P \Rightarrow I'$. In summary, there is no need in considering state invariants during the "Generation" step as long as the corresponding proof obligations have been discharged. Therefore, in Fastest we encourage to write test invariants outside the state schema.

III. On-Board Flight Control Software of a Satellite Launcher Vehicle

We have written a Z model describing how the on-board software of a satellite launcher vehicle shall control the events sent by different sensors, called *time reference events* (TRE)—the requirements were provided by researchers from Instituto de Aeronáutica e Espaço (IAE), Brazil. These events are used to calculate when the *relative events* shall be issued by the control software. The Z model covers only those TRE that control lift-off and thrust drop detection (three different events). This is the smallest and simplest model reported in this paper because it encompasses only 139 lines of Z LaTeX markup. The basic elements of the model are the following.

$$STATUS ::= normal \mid failure$$

$$\begin{aligned} TRE ::= &\\ & LiftOff \\ & \mid ThrustDrop1E \\ & \mid ThrustDrop2E \\ & \mid ThrustDrop3E \end{aligned}$$

$tli\ e$ ($tls\ e$) stands for lower (upper) time bound of reference event e. That is, TRE cannot be received at any time but within specified time frames. Similarly, $X\ e$ is the acceleration level defined for detection of reference event e.

$$tli, tls, X : TRE \rightarrow \mathbb{N}$$

The following schema represent the state space of the system.

```
OBS
```
$$\begin{array}{|l} now : \mathbb{N} \\ ot : TRE \nrightarrow \mathbb{N} \\ sysState : STATUS \\ fa : \mathbb{N} \end{array}$$

The meaning of the state variables is as follows:

Current flight time $\approx now$
Occurrence time of reference event $e \approx ot\ e$
Current system state $\approx sysState$
Current vehicle acceleration as informed by the appropriate sensor $\approx fa$

There is one set of operation schemas describing what to do when each TRE is detected; these schemas are further subdivided describing successful and erroneous situations. For instance, the following schemas describe the detection of the lift-off event (some irrelevant predicates are omitted):

DetectLiftOffOk
$$\begin{array}{|l} \Delta OBS;\ e? : TRE \\ \hline e? = LiftOff \\ sysState = normal \\ e? \notin \operatorname{dom} ot \\ now \in tli\ e? \mathinner{..} tls\ e? \\ X\ e? \leq fa \\ ot' = ot \cup \{e? \mapsto now\} \\ \ldots \end{array}$$

EventOutOfTF
$$\begin{array}{|l} \Delta OBS;\ e? : TRE \\ \hline sysState = normal \\ now \notin tli\ e? \mathinner{..} tls\ e? \\ sysState' = failure \\ \ldots \end{array}$$

$$\begin{array}{|l}
\textit{DetectLiftOffAccelError} \\
\Delta OBS;\ e? : TRE \\
\hline
e? = \textit{LiftOff} \\
\textit{sysState} = \textit{normal} \\
X\ e? > \textit{fa} \\
\textit{sysState}' = \textit{failure} \\
\ldots \\
\hline
\end{array}$$

$$\textit{DetectLiftOffOOTF} == [\textit{EventOutOfTF} \mid e? = \textit{LiftOff}]$$

$$\begin{array}{l}
\textit{DetectLiftOff} == \\
\quad \textit{DetectLiftOffOk} \\
\quad \lor \textit{DetectLiftOffAccelError} \lor \textit{DetectLiftOffOOTF}
\end{array}$$

We were able to automatically generate abstract test cases for testing the detection of all the events in different time frames, as is shown in Table I. Column **Tactics** lists the testing tactics that we applied for each operation—we omit to mention DNF because Fastest always applies it. Columns **Leaves** and **Pruned** show the initial number of leaves of the testing tree and the number of leaves automatically pruned by Fastest, respectively. **Auto**, **Manual** and **Total** are the number of abstract test cases automatically generated by the tool, those that needed manual intervention, and the sum of them, respectively. Since a test case is tuple of constants satisfying the predicate of the test class, there is no algorithm that can find all of them [6]. Therefore, the user must help the algorithm so it can generate some test cases. The **Manual** column records the number of test cases that required the user to assist the algorithm in finding a test case.

A. Rationale Behind Tactic Selection

In this section we want to explain why we applied DNF, SP and NR to *DetectLiftOff* to show the rationale behind the testing tactic selection process. Applying DNF means to generate a test class for each of schemas that form *DetectLiftOff*. This implies that at least one test case satisfying the precondition of each schema will be generated. Since the schemas capture right and unexpected behaviours of the operation, DNF guarantees to test the implementation in all these situations.

After DNF, we applied SP to the $\cup$ operator appearing in the postcondition of *DetectLiftOffOk*—recall SP from Section II at page 2 and Figure 2. As we explained in Section II, SP generates test to exercise the program with different values of ot in a way that the implementation of $ot' = ot \cup \{e? \mapsto now\}$ is tested quite deeply.

By applying NR to *now* (the state variable measuring the current time) Fastest generates tests for the *LiftOff* event arriving at different moments with respect to the time frame in which the event is waited—recall NR from Section II-A. Therefore, we asked Fastest to generate test classes requiring that *now* varies over these time frames. First, time frames are defined by giving values to the axiomatic definitions, *tli* and *tls*—the values for *tli* and *tls* were suggested by a domain expert from IAE, but they are fictitious. Second, the NR tactic is applied by setting the lists of numbers to the values given in the previous step.

Below we show a typical test class and its test case.

$$\begin{array}{|l}
\textit{DetectLiftOff\_NR\_}18 \\
\textit{DetectReferenceEvent\_VIS} \\
\hline
e? = \textit{LiftOff} \\
\textit{sysState} = \textit{normal} \\
e? \notin \operatorname{dom} ot \\
now \in tli\ e? \mathrel{..} tls\ e? \\
X\ e? \leq \textit{fa} \\
ot \neq \varnothing \\
ot \cap \{e? \mapsto now\} = \varnothing \\
1 < now < 3 \\
\hline
\end{array}$$

$$\begin{array}{|l}
\textit{DetectLiftOff\_NR\_}18\textit{\_TCASE} \\
\textit{DetectLiftOff\_NR\_}18 \\
\hline
ot = \{(\textit{ThrustDrop}1E, 57)\} \\
e? = \textit{LiftOff} \\
\textit{sysState} = \textit{normal} \\
\textit{fa} = 57 \\
now = 2 \\
\hline
\end{array}$$

Although more abstract test cases could have been generated by applying more testing tactics—for example, to test the arrival of TRE at different accelerations—, we believe the ones obtained so far would exercise the implementation in the most relevant ways.

IV. Two Satellite Communication Protocols

In this section we present two closely related problems regarding the formalization of communication protocols implemented in satellites developed by Instituto Nacional de Pesquisas Espaciais (INPE) from Brazil. In a joint project between CIFASIS, Flowgate and INPE, researches from INPE provided the protocol requirements in Portuguese and researchers from CIFASIS and Flowgate wrote the Z specifications and applied Fastest. These results were compared with those produced by another MBT technique applied at INPE [20]. Here we show the application of the TTF and Fastest with greater detail.

These protocols describe the messages that two on-board computers of a satellite can exchange during the mission. In both cases there is a dedicated computer that communicates with the On-Board Data Handling (OBDH) computer, the satellite platform computer that processes, generates and formats platform and payload information that is transmitted to ground stations. Our Z specifications correspond to the software running on each dedicated computer. These specifications are structured as a set of Z schemas each describing what the software must do when a particular message from OBDH arrives, plus some schemas for internal responses to previous messages. All the operations of these models include schemas for the normal and erroneous situations.

Operation	Tactics	Leaves	Pruned	Auto	Manual	Total
DetectLiftOff	SP $\cup$, NR *now* $\langle 1, 3\rangle$	46	37	8	1	9
*DetectThrustDropE*1	SP $\cup$, NR *now* $\langle 10, 15\rangle$	46	37	8	1	9
*DetectThrustDropE*2	SP $\cup$, NR *now* $\langle 27, 30\rangle$	46	37	6	3	9
*DetectThrustDropE*3	SP $\cup$, NR *now* $\langle 50, 56\rangle$	46	37	6	3	9
Totals		184	141	28	8	36

TABLE I
MBT APPLIED TO THE DETECTION OF EVENTS OF A SATELLITE LAUNCHER VEHICLE.

A. EXP-OBDH Communication Protocol

In this case the dedicated computer, called EXP, performs astrophysical experiments. This specification comprises the following operations:

- The reception of commands from OBDH, including a real-time requirement.
- The acquisition of scientific data sensed by the astrophysical devices.
- An array of simple operations fired by commands sent from OBDH: reset the micro controller, return the time measured by the clock of the EXP computer, initiate and stop data acquisition, and so on.
- A simple internal operation to interrupt the telescope every 700 ms so it can interrupt EXP when data is available.
- Load the memory containing experimental data with data sent from OBDH.
- Transmission of experimental data to OBDH.

Note the combination of both reactive behaviour and complex data structures; this is recurring in the rest of the cases studies presented in this paper.

The specification comprises ten pages with 608 lines of Z code. The state space of the model declares, among others, a variable representing EXP's memory area reserved for experimental data. This variable, called *memd*, is modified when the telescope interrupts EXP to copy scientific data and when a memory dump is requested. We specified the first operation as follows:

```
RetrieveExpDataOk
  $\Delta ExpState;\ \Xi Time;\ \Xi Status;\ data? : \operatorname{seq} MDATA$
  ----
  $data? \neq \langle\rangle$
  $mep + \#data? \leq 43$
  $memd' =$
      $memd \oplus \{i : 1 \mathrel{..} \#data? \bullet mep + i \mapsto data?\ i\}$
  $mep' = mep + \#data?$
  $ped' = 0$
  $\dots$
```

$$RetrieveExpDataE == [\Xi ExpState;\ data? : \operatorname{seq} MDATA \mid data? = \langle\rangle \lor 43 < mep + \#data?]$$
$$RetrieveExpData == RetrieveExpDataOk \lor RetrieveExpDataE$$

data? receives the data sent by the telescope; it must be a non empty sequence. *mep* is a pointer pointing to the last used memory cell in *memd*. Therefore, we include a precondition avoiding a buffer overflow. If these preconditions are met, the memory is updated by overwriting *memd* with *data*? starting from *mep*; and *mep* is incremented according to the length of *data*?.

After loading the model into Fastest we applied DNF and SP to the $\oplus$ operator. Applying DNF warrants to test at least the successful and the erroneous cases, one of which includes testing for a buffer overflow. Applying SP, tests the implementation of the copy routine with significant values of *memd*, *data*? and *mep*. For instance, Fastest produces, among others, test classes equivalent to the following:

$$memd = \langle\rangle \land data? \neq \langle\rangle \land mep = 0$$
$$\#memd \geq 2 \land \#data? = 1 \land mep = 0$$
$$\#memd = \#data? = 1 \land mep = 1$$

These can be translated into English, respectively, as follows: 'it is the first time the telescope sends data'; 'the telescope sends data after a memory dump was requested (*memd* is non empty but *mep* points to the initial address)'; and 'is not the first time the telescope sends data'.

Depending on the mathematics involved in each Z operation we tested all of them following the same criteria. Table II summarizes the testing tactics and testing results of this case study. The last row shows that we generated 86 abstract test cases for the whole specification. As with the other case studies, more could have been generated by applying more testing tactics, but these exercises the main alternatives described by the model.

B. SWPDC

SWPD is the software embedded into the Payload Data Handling Computer (PDC). Although similar in conception to the previous case study, SWPDC is much more complex because it handles not only scientific data but also dump data (e.g. housekeeping data), accomplishes data memory management, implements flow control mechanisms, etc. Data transmission is more complex since SWPDC has to keep record of the last transmitted frame because OBDH can ask it again if some problem during transmission was detected. This increased complexity is reflected in the number of lines of Z code which more than doubles that of the previous case study.

SWPDC includes four memory pages which are relevant to the formal model. These pages are of 32 Kb each. On the other hand, some requirements define the size of the data frames that are transmitted back and forth between PDC and OBDH—1,111 and 1,116 bytes. The model we have written

Operation	Tactics	Leaves	Pruned	Auto	Manual	Total
CommandFinish		2	0	2	0	2
CommandStart		2	0	2	0	2
CommandType	FT $type?$	28	18	10	0	10
InitDataAcq		4	0	4	0	4
LoadParam	FT $p?$	13	0	13	0	13
MemoryDump		3	0	3	0	3
MemoryLoad	SP $\oplus$	21	4	15	2	17
Reconfig	FT $nmode?$	4	0	4	0	4
ResetMicro		3	0	3	0	3
RetrieveEData	SP $\oplus$	10	2	8	0	8
SendClock		3	0	3	0	3
StopDataAcq		4	0	4	0	4
TransData	SP $\lhd$	14	1	9	4	13
Totals		111	25	80	6	86

TABLE II
MBT APPLIED TO THE EXP-OBDH COMMUNICATION PROTOCOL.

includes a detailed description of memory management and data transmission, which needs references to these constants. Then, some abstract test cases would need to take into account these constants. For instance, there would be abstract test cases to test whether overflowing each memory page is possible or not. This, in turn, would require to write a list of, say, 32,767 elements representing the current state of one of those memory pages. Since this is very difficult to accomplish automatically and even manually, we changed these constants for smaller numbers. It remains a challenge to automatize the generation of such large constants values.

As Table III shows, the results are rather similar to those of the previous case study. To generate test cases for the most complex operation, *TransmitData*, we applied SP to two different expressions and then we applied NR, again, to two different variables to consider possible buffer overflows. As can be seen the initial number of test classes generated for this operation is quite high, but many of them are pruned since the operation has a number of restrictions. One of these restrictions makes it necessary to add new elimination theorems to detect contradictions coming from total functions. For example, the following elimination theorem codifies a contradiction only valid for total functions:

$$\textbf{ETheorem}\ \mathrm{T1}\ [f : X \rightarrow Y;\ a : X;\ b : Y]$$
$$\mathrm{dom} f \cap \mathrm{dom}\{a \mapsto b\} = \varnothing$$

This elimination theorem prunes test classes generated by applying SP to an expression of the form $f \oplus \{x \mapsto y\}$ where f is a total function, since one of the partitions defined by SP for $R \oplus G$ is:

$$R \neq \varnothing, G \neq \varnothing, \mathrm{dom}\, R \cap \mathrm{dom}\, G = \varnothing;$$

V. MBT of the ECSS-E-70-41A Standard

We have developed a Z specification of a significant portion of the ECSS-E-70-41A Aerospace Standard [16]. This is the largest model we have written so far, including complex Z concepts. It posed a number of theoretical and practical challenges, that are the subject of current efforts for improving the TTF and Fastest. In this section we report on the MBT activities we were able to do with this model and succinctly describe some of the problems we faced.

A. A Complex Specification

We have formalized the minimum capability sets of the following services[2]:

- Telecommand verification service: Telecommand Acceptance Report Success (1,1); Telecommand Acceptance Report Failure (1,2).
- Housekeeping and diagnostic data reporting service: Housekeeping Parameter Report (3,25); Define New Housekeeping Parameter Report (3,1); Clear Housekeeping Parameter Report Definitions (3,3); Enable Housekeeping Parameter Report Generation (3,5); Disable Housekeeping Parameter Report Generation (3,6).
- Memory management service: Load Memory using Absolute Addresses (AA) (6,2); Dump Memory using AA (6,5); Memory Dump using AA Report (6,6).
- On-board monitoring service: Enable Monitoring of Parameters (12,1); Disable Monitoring of Parameters (12,2); Check Transition Report (12,12).
- On-board storage and retrieval: Enable Storage in Packet Stores (15,1); Disable Storage in Packet Stores (15,2); Packet Store Contents Report (15,8); Downlink Packet Store Contents for Time Period (15,9).

As the list shows, we have formalized 17 service subtypes for 6 of the 16 services described in the standard. We chose the services on the basis of their functional impact or richness, their independence from specific missions and the potential complexity of their implementations. For instance, since the functionality of the "Test service" is mostly mission-specific, the "Device command distribution service" is a low-level service with very limited functionality, and the "Function management service" is a rather simple function-call dispatching service, we decided not to formalize them. Although we decided to exclude some services, others may be good candidates to be included in the model.

The specification has 2,011 lines of Z LaTeX markup in a 74 pages document; including 25 state variables with 16 of a

[2]The ordered pairs are the standard identifiers for services in [16]. The first component identifies the *service type* and the second the *service subtype*; we will use them to shorten the presentation.

Operation	Tactics	Leaves	Pruned	Auto	Manual	Total
ChangeOperationMode	FT $m?$	5	0	5	0	5
CmdAccept		3	0	3	0	3
ExecuteProgram		5	0	5	0	5
GetMemoryDumpReady	ISE $p? \in \{0, \dots, 7\}$	13	0	13	0	13
LoadMemory	SP $\oplus$	94	62	17	7	24
ReceiveHardwareSignal	FT $s?$	9	0	9	0	9
ModSoftParam	SP $\oplus$	31	18	13	0	13
ReinitDataAcquisition		4	0	4	0	4
ResendsAnswer		3	0	3	0	3
SendClock		3	0	3	0	3
StopDataAcquisition		4	0	4	0	4
TransmitData	SP $\oplus$ (2) NR *sdwp ia*	73	54	14	5	19
VerifyOperationMode		5	0	5	0	5
Totals		252	134	97	12	109

TABLE III
MBT APPLIED TO THE SOFTWARE EMBEDDED INTO THE PDC.

relational type, of which 6 are higher-order functions and 3 are defined by referencing schema types. This should give an idea of the actual complexity of the model[3]. For instance, the following state variable:

$$\begin{aligned} &hSampVal : \\ &\quad SID \nrightarrow \\ &\qquad (PARAMNAME \nrightarrow (TIME \nrightarrow PARAMVALUE)) \end{aligned}$$

is used to record the sampling of different housekeeping parameters at different times, according to the definition of different report definitions; while:

$$hRepDef : SID \nrightarrow HRepDef$$

records the report definitions themselves, where *HRepDef* is a schema type having 6 fields (not counted within the state variables), 3 of which are partial functions.

Since many state variables are unrelated to each other, we factorized out the model state schema in one schema per service. The formalization of each subtype is essentially a Z operation which in turn may comprise several schemas. In this way, schemas formalizing a subtype of a given service only access and modify the state schema corresponding to that service. The specification of any subtype comprises both the normal and erroneous conditions.

Another issue contributing to the complexity of the model is that it contains 28 axiomatic definitions. Some of these are even more complex than the one shown in Figure 4, which formalizes the notion of a parameter consistently failing a low-limit check as required by the on-board monitoring service—note that *consistFailedLowL* also references another complex axiomatic definition, *lastRepVal*. This is our first model featuring axiomatic definitions of such complexity.

B. Deriving Test Classes and Beyond

Although our ECSS-E-70-41A formalization includes Z features not covered by the TTF and not yet implemented by Fastest (see Section V-C), we were able to use it to produce some interesting results, as described in the following subsections.

1) Generating Test Classes for the ECSS-E-70-41A Standard.: The most significant result we were able to obtain regarding this model is a set of 256 test conditions for some of the services, as shown in Table IV[4]. These test classes can be used to guide the functional testing of any implementation. It would be possible to generate many more test conditions and test cases by applying more tactics, or by developing improvements to the TTF and Fastest (see Section V-C). For instance, we applied tactics mostly to generate test classes for those schemas specifying the successful execution of an operation, as we did in the example of Section II. Therefore, one way of generating more test classes would be by applying testing tactics to the schemas representing errors.

Test classes for (3,1) include those derived for a promoted operation that does the hard work of saving the new report definition (*NewHRepDefOk*). In (15,1) we have included test classes derived for the internal operation that is responsible for saving the packets in the packet stores (*SavePackets*).

The services that were most affected by the limitations of the TTF and Fastest are (3,25), (6,2), (6,6), (12,1), (12,2), (12,12), (15,1) and (15,9). They use language expressions that we cannot currently consider to partition their VIS and that are likely to have complex implementations. Particularly, for (3,25) we were unable to generate test classes since it uses many Z features not yet covered by our methodology. Another example is *SavePackets* since the specification of this operation is based essentially on **let** expressions which made Fastest to produce an incomplete DNF calculation.

2) A Case Study Using the Formal Model.: INVAP is a satellite developer from Argentina that wanted to assess an MBT method and decide whether it would be useful as a replacement for their manual process. They routinely base their developments on the ECSS standards suite. We therefore proposed using our ECSS-E-70-41A formalization and Fastest

[3]As a matter of comparison, the Tokeneer specification has only 46 lines more, and it is widely recognized as a full-fledged, industrial-strength formal specification [21].

[4]Here column **Leaves** is the number of test classes after pruning.

$$
\begin{array}{|l}
consistFailedLowL : \mathbb{P}((TIME \pfun PARAMVALUE) \times PARAMVALUE \times CheckDef) \\
\hline
\forall h : TIME \pfun PARAMVALUE;\ v : PARAMVALUE;\ d : CheckDef \bullet \\
\quad (h, v, d) \in consistFailedLowL \Leftrightarrow max(\mathrm{ran}(lastRepVal\ h\ v\ d.rep)) < d.low
\end{array}
$$

Fig. 4. The formalization of ECSS-E-70-41A includes some axiomatic definitions more complex that the one presented here.

Op/Srv	Tactics	Leaves	Op/Srv	Tactics	Leaves
(1,1) & (1,2)		4	(6,6)		2
(3,1)	SP $\cup$ (2), $\cap$, $\notin$	38	(12,1)	SP $\oplus$	5
(3,3)	SP $\lhd$ (2), $\setminus$	75	(12,2)	SP $\oplus$	5
(3,5)	SP $\cup$ (2)	28	(12,12)		2
(3,6)	SP $\setminus$, *ndres*	22	(15,1)	SP $\oplus$	15
(3,25)		–	(15,2)	SP $\oplus$	12
(6,2)	SP $\#$, MTS *sa?*, *len?*	16	(15,8)		3
(6,5)	SP $\#$, MTS *sa?*, *len?*	16	(15,9)	SP $\in$ (4)	12

TABLE IV
TEST CLASSES DERIVED FOR ECSS-E-70-41A'S SERVICES.

to generate abstract test cases for some typical subtypes, namely, (1,1), (1,2), (6,5), (6,6), (12,1) and (12,2). As part of the case study, they also needed an English description of each test case, which are used as evidence for external validators. INVAP had already derived test cases for the agreed services but we did not know them in advance, and, at the same time, we had written the model some time before. However, we had to modify it slightly to be able to generate test cases, and not only test classes. Modifications involved performing some schema expansions that Fastest is not able to do so far.

Table V shows a summary of the results of the case study. By applying the TTF and Fastest, we were able to generate roughly the same test cases derived manually by the company. Also, by borrowing a rather simple template-based method from the natural language generation community [22], we were able to semi-automatically translate the Z test cases into English as requested by the company [23]. Clearly, this is a surplus of the MBT approach, as formal test cases can serve more purposes than those generated manually. The company was satisfied with these results, although they would like to have the expected output of each test case, as they regard test cases as (*provided input, expected output*) pairs. The TTF, on the other hand, gives a final answer in terms of the presence or absence of an error in the implementation, rather than including outputs in test cases.

C. Challenges and Open Issues

As we have said, this model posed a number of challenges, mostly due to the inclusion of complex Z concepts. These concepts were not originally addressed by the TTF and are consequently not supported by Fastest. Therefore, the first step would be to extend the TTF, so that a possible implementation can be foreseen. The issues we will focus on are the following:

– Lack of tactics to deal with λ expressions and set comprehensions. This is, nevertheless, rather easy to deal with because in Z any λ expression can be written as a set comprehension and these, in turn, are essentially first order predicates. The TTF is well equipped to work with predicates.
– Lack of tactics to work with higher order functions. It would be interesting to develop tactics to partition higher order functions. Although a higher order function is a function, they have some peculiarities that should be explored.
– Schema types. Schema types are strongly related to hierarchical specifications where low-level operations are promoted to higher level state machines. In our opinion hierarchical specifications are tightly related to integration testing due to the clear mapping between low and high level operations and low and high service layers of the implementation. This, in turn, implies to study how to use MBT for integration testing.
– Support for variable hiding, **let** expressions, and quantifications need to be improved.
– Poor support for axiomatic definitions. Actually the TTF does not say anything about this important Z feature. However, Fastest includes a limited support. We treat them as model parameters that need to be bound to a constant value before test cases are derived. This is a good approach, for instance, for configuration variables. On the other hand, binding a value to, say, *consistFailedLowL* (shown above) might be unnecessary. In this case, the variable is equivalent to a predicate that can be replaced whenever it is referenced. We need to study a more general solution for this issue.
– Generics. Fastest does not support generic definitions, generic schemas, and the like, but: (a) some of them are used mostly to extend the notation rather than to write specifications; and (b) some testing tactics can be applied to their "uses" instead to their definitions.

Our current efforts are oriented towards solving these and other minor issues, both in their theoretical and practical

Operation	Tactics	Leaves	Pruned	Auto	Manual	Total
TelecmdAccept		4	0	4	0	4
DumpMemoryAbsAdd	SP $\leq$	32	18	7	7	14
MemoryDumpAbsAdd	SP $\upharpoonright$	15	4	9	2	11
EnableMonitParam	SP $\oplus$, FT *srv*	250	172	50	28	78
DisableMonitParam	SP $\oplus$, FT *srv*	250	172	50	28	78
Totals		551	366	120	65	185

TABLE V
TEST CASES DERIVED FOR SOME ECSS-E-70-41A'S SERVICES.

aspects.

VI. CONCLUSIONS

We have applied a particular MBT theory with tool support to four case studies from the aerospace domain. They range from rather simple models of aerospace systems to a highly complex formalization of an aerospace standard. Our first conclusion is that the Z formal notation is adequate to formalize aerospace software, where reactive behaviour and complex data structures are combined to form nontrivial infinite-state machines. From this kind of models, the TTF permits to generate detailed test cases by applying simple testing tactics that easily consider all the functional alternatives codified in the model, thus guaranteeing a very good coverage of the implementation. By applying Fastest, a faithful implementation of the TTF, we were able to automatically generate dozens of test cases, turning the testing process into analytical work rather than a tedious manual task.

ACKNOWLEDGMENTS.

We would like to thank the following people and institutions for their support in writing this paper. Miriam Alves from IAE, Valdivino Alexandre from INPE, Flowgate Consulting, REVVIS, INVAP and LSIS.

REFERENCES

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

[2] R. M. Hierons, K. Bogdanov, J. P. Bowen, R. Cleaveland, J. Derrick, J. Dick, M. Gheorghe, M. Harman, K. Kapoor, P. Krause, G. Lüttgen, A. J. H. Simons, S. Vilkomir, M. R. Woodward, and H. Zedan, "Using formal specifications to support testing," *ACM Comput. Surv.*, vol. 41, no. 2, pp. 1–76, 2009.

[3] P. Stocks and D. Carrington, "A Framework for Specification-Based Testing," *IEEE Transactions on Software Engineering*, vol. 22, no. 11, pp. 777–793, Nov. 1996.

[4] P. Stocks, "Applying formal methods to software testing," Ph.D. dissertation, Department of Computer Science, University of Queensland, 1993.

[5] I. MacColl and D. Carrington, "Extending the Test Template Framework," in *Proceedings of the Third Northern Formal Methods Workshop*, Ilkely, UK, 1998.

[6] M. Cristiá and P. Rodríguez Monetti, "Implementing and applying the Stocks-Carrington framework for model-based testing," in *ICFEM*, ser. Lecture Notes in Computer Science, K. Breitman and A. Cavalcanti, Eds., vol. 5885. Springer, 2009, pp. 167–185.

[7] M. Cristiá, P. Albertengo, and P. Rodríguez Monetti, "Pruning testing trees in the Test Template Framework by detecting mathematical contradictions," in *SEFM*, J. L. Fiadeiro and S. Gnesi, Eds. IEEE Computer Society, 2010, pp. 268–277.

[8] M. Cristiá, P. Albertengo, and P. Rodríguez Monetti, "Fastest: a model-based testing tool for the Z notation," in *PTD-SEFM*, F. Mazzanti and G. Trentani, Eds. Consiglio Nazionale della Ricerche, Pisa, Italy, 2010, pp. 3–8.

[9] The Eclipse Foundation. Eclipse. [Online]. Available: http://www.eclipse.org/

[10] T. Hupponen, K. Karlsson, J. Laitinen, O. Ojala, A. Pirinen, E. Seuranen, and L. Takkinen. TeXlipse. [Online]. Available: http://texlipse.sourceforge.net/

[11] CZT. CZT Eclipse Plugin. [Online]. Available: http://www.cs.waikato.ac.nz/~marku/czt/eclipse.html

[12] M. Cristiá, P. Rodríguez Monetti, and P. Albertengo, "The Fastest 1.3.6 User's Guide," Flowgate Consulting, Tech. Rep., 2010. [Online]. Available: http://www.flowgate.net/pdf/userGuide.pdf

[13] Maximiliano Cristiá. Fastest. [Online]. Available: http://www.fceia.unr.edu.ar/~mcristia

[14] J. M. Spivey, *The Z notation: a reference manual*. Hertfordshire, UK, UK: Prentice Hall International (UK) Ltd., 1992.

[15] B. Potter, D. Till, and J. Sinclair, *An introduction to formal specification and Z*. Prentice Hall PTR Upper Saddle River, NJ, USA, 1996.

[16] ECSS, "Space Engineering – Ground Systems and Operations: Telemetry and Telecommand Packet Utilization," European Space Agency, Tech. Rep. ECSS-E-70-41A, 2003.

[17] P. Malik and M. Utting, "CZT: A framework for Z tools," in *ZB*, ser. Lecture Notes in Computer Science, H. Treharne, S. King, M. C. Henson, and S. A. Schneider, Eds., vol. 3455. Springer, 2005, pp. 65–84.

[18] ISO, "Information Technology – Z Formal Specification Notation – Syntax, Type System and Semantics," International Organization for Standardization, Tech. Rep. ISO/IEC 13568, 2002.

[19] M. Cristiá, D. Hollmann, P. Albertengo, C. Frydman, and P. Rodríguez Monetti, "A language for test case refinement in the Test Template Framework," in *ICFEM*, 2011, p. to appear.

[20] M. Cristiá, V. Santiago, and N. Vijaykumar, "On comparing and complementing two MBT approaches," in *LATW*, F. Vargas and E. Cota, Eds. IEEE Computer Society, 2010, pp. 1–6.

[21] J. Barnes, R. Chapman, R. Johnson, J. Widmaier, D. Cooper, and B. Everett, "Engineering the Tokeneer enclave protection software," in *Proceedings of the IEEE International Symposium on Secure Software Engineering*. IEEE, 2006.

[22] E. Reiter and R. Dale, *Building Natural Language Generation Systems*. Cambridge, UK: Cambridge University Press, 2000.

[23] M. Cristiá and B. Plüss, "Generating natural language descriptions of z test cases," in *INLG*, J. D. Kelleher, B. M. Namee, I. van der Sluis, A. Belz, A. Gatt, and A. Koller, Eds. The Association for Computer Linguistics, 2010.