A Functional Verification of a Web Voting System

Maximiliano Cristiá[1] and Claudia Frydman[2]

[1] CIFASIS and UNR, Rosario, Argentina
cristia@cifasis-conicet.gov.ar
[2] LSIS-CIFASIS, Marseille, France
claudia.frydman@lsis.org

Abstract. The Consejo Nacional de Investigaciones Científicas y Técnicas (CONICET) is the most important research institution in Argentina. Its internal authorities are elected by around 8,000 researches across the country. During 2011 the CONICET developed a web voting system to replace the traditional mail-based system. In this paper we present the verification process conducted to assess the functional correctness of the voting system. This process is the result of integrating automatic and semi-automatic verification activities from formal proof to code inspection and model-based testing.

1 Introduction

The Consejo Nacional de Investigaciones Científicas y Técnicas (CONICET) is the most important research institution in Argentina. Some of its internal authorities, including some of the members of its Board of Directors, are elected by some of the more of 8,000 CONICET's researchers. Traditionally, the election process was carried out manually. In 2011 the Board of Directors commissioned the development of an electronic voting system. Eventually, it was decided that CONICET's Systems Department would develop a web-based system.

The Board of Directors required the Systems Department to subject the system to an external evaluation to determine whether it fulfilled functional, security and availability quality attributes, before moving it into production. Although the program is not large it uses some complex technologies and a failure may have a high political impact. However, the time frame assigned to this activity was only one calendar month. Since the external evaluators could not have any political interests in the system, the System Department contacted us to perform the evaluation. We conducted the functional evaluation, and two security researchers performed the security and availability evaluation. This paper is an experience report on the verification activities conducted during the *functional* evaluation of the voting system. It is important to remark two points: a) we are reporting only on the functional verification which means that we do not make any comments about security, although this attribute is important for the overall quality of the system; and b) the aim of the task was to assess the quality of an existing system and not to develop it from scratch.

Given the potential risk of a functional error we decided to conduct an evaluation as formal as possible within the available time frame (one month). So, the first step was to write a formal specification of the set of regulations governing the election process issued by the Board of Directors (Section 2). Secondly, we formally proved that the specification verifies some state invariants as a way to have some confidence that the model is correct (Section 3). The third step consisted in inspecting the source code to find each pre and postcondition of each operation of the model, and annotating the source code with links to the corresponding predicates of the formal specification (Section 4). In the last step we applied Fastest, a model-based testing (MBT) tool, to generate test cases (Section 5). In this way, the formal specification written in the first step was used in all the latter activities. In the rest of the paper we further discuss these and other issues (Section 6) and survey some related papers (Section 7); our conclusions are in Section 8. This paper is a reduced version of a technical report including the full Z specification and the Z/EVES proof scripts. It can be downloaded from `https://www.dropbox.com/s/c6bawikdrd608c9/votingSystem.tar.gz`.

It is worth noticing that this project presents two important differences with respect to other verification efforts that are reported in the academic community. Firstly, in most other reports on verified software the development team can either write a formal specification of the system and then develop the implementation using many different techniques, or the verification consist on proving some non-functional properties of a given implementation—for instance, code-level safety properties such as memory safety. In this project, however, the implementation was already developed and a full *functional* verification was requested. Therefore, we were not allowed to generate the implementation we liked, and we had to go beyond of code-level safety properties. Secondly, the implementation was developed by average programmers, and not, for instance, by people holding a PhD in software engineering, formal methods or formal verification. This is the case for almost all the software produced in the world.

On June, 15, 2012 the first web-based election of two members of the Board of Directors was carried out without any noticeable disruption or failure of the new election system.

2 From Requirements to Formal Specification

In this project the functional requirements are the regulations set by CONICET for the election process. This document is essentially a legal document written in Spanish stipulating all the conditions for an election. It is divided in sections (for example, “On the Voters”, “On the Candidates to Be Elected”, etc.) and each section is organized as a list of articles or clauses. We derived a formal specification for the system from this document plus just a few questions to administrative personnel of CONICET (i.e. no intermediate representation of the requirements was developed).

We wrote a formal functional specification as the fundamental cornerstone of the functional verification process. We knew in advance that we could use it

for many different verification activities. The Z notation [31, 21] was the chosen language because: a) we are fluent in it and some of its tools; b) Z is a very good notation to formalize this kind of requirements; and c) our research and tools on MBT [10] would be of great help (and conversely, this would become a case study for our investigations).

The Z specification resulting from the requirements document is a rather standard Z model although we introduced two style changes. First, we used a form of Jackson's designations [22] as a means of linking the formal model with the requirements. For instance, before introducing the set representing CONICET's researchers we provide the following designation:

$$x \text{ is an active or retired researcher of CONICET} \approx x \in RSCH$$

so, then, we introduce the following Z paragraph:

$$[RSCH]$$

As can be seen, the left hand side of a designation is an informal sentence over the requirements while the right hand side is a formal term of the model.

After declaring types representing the main concepts involved in the election rules, we can give the main components of the state of the system. *Endorsements* records preliminary candidates and their endorsements, while *Votes* records firm candidates and voting data.

Endorsements

$$cand : RSCH \pfun KA$$
$$endors : RSCH \pfun \mathbb{F}\, RSCH$$

Votes

$$firmCand : \mathbb{P}\, RSCH$$
$$voted : \mathbb{F}\, RSCH$$
$$votes : \mathrm{seq}(\mathbb{F}\, RSCH)$$

The second change in style introduced in this model concerns state invariants. As can be seen, we have not included state invariants in the state schemas as is customary in Z. Rather, we write them in a separate schema as shown in Fig. 1. Some of the predicates in *Invariants* are the formalization of CONICET rules. For example, CONICET established that after the end of the period reserved for endorsements (*fEndors*) only candidates who have got at least 20 endorsements become firm candidates (see the seventh predicate in *Invariants*)[3]. In a sense, this way of writing invariants changes "calculate the full precondition of an operation" by "discharge a proof obligation" [30, 10]. Precisely, to be sure that these predicates are indeed invariants, we include a proof obligation such as the following, for each operation in the model (*Vote* is one of the operations):

theorem VotePreservesInvariants
$$Invariants \land Vote \Rightarrow Invariants'$$

In other words, if this theorem cannot be proved it means that operation *Vote* does not verify one of the expected properties of the model. The same holds for all the invariants and operations.

[3] A complete description of the invariants can be found in the technical report associated to this paper, as mentioned in the introduction.

Invariants

$Endorsements;\ Votes;\ Time$

$cand \subseteq CanBeCandidates$
$\mathrm{dom}\ endors = \mathrm{dom}\ cand$
$\bigcup(\mathrm{ran}\ endors) \subseteq CanVote$
$\forall i : \mathrm{dom}\ endors \bullet endors\ i \neq \emptyset \Rightarrow \{myKA\ i\} = myKA(\!|endors\ i|\!)$
$\forall i, j : \mathrm{dom}\ endors \mid i \neq j \bullet endors\ i \cap endors\ j = \emptyset$
$today \leq fEndors \Rightarrow firmCand = \emptyset$
$fEndors < today \Rightarrow firmCand = \mathrm{dom}(endors \rhd \{A : \mathrm{ran}\ endors | 20 \leq \#A\})$
$today < bVote \Rightarrow voted = \emptyset \land votes = \langle\rangle$
$\bigcup(\mathrm{ran}\ votes) \subseteq firmCand$
$\#votes = \#voted$
$voted \subseteq CanVote$

Fig. 1. States invariants for CONICET's election system.

We close this section by introducing the operation representing a voter issuing his or her vote with the following schema expression:

$$
\begin{aligned}
&Vote == \\
&\quad VoteOk \\
&\quad \lor\ VoteWrongDate \lor ResearcherCannotVote \\
&\quad \lor\ AlreadyVoted \lor VoteMoreThanThree \\
&\quad \lor\ VoteNonCandidates \lor KACandidatesIsWrong \\
&\quad \lor\ GRCandidatesIsWrong
\end{aligned}
$$

where *VoteOk*, shown in Fig. 2, formalizes the situation when a voter successfully issues his or her vote; all the other schemas describe possible errors. The labels written to the right of the schemas, such as $\mathsf{Pre} - 1$, will be used to identify each predicate as is explained in Sect. 4. In *VoteOk*, $e?$ is the researcher issuing the vote and $C?$ is his or her vote. According to CONICET rules a vote may contain up to three candidates ($\#C? \leq 3$). Below we include one of the "error" schemas just to illustrate them:

$$AlreadyVoted == [\Xi ES;\ e? : RSCH \mid e? \in voted \qquad \mathsf{Pos} - 3]$$

3 Proving Properties of the Specification

The Z specification was verified under the Z/EVES system [30]. Z/EVES generates a proof obligation, called domain check, every time a partial function is applied to an argument. The proof obligation asks to prove that the argument belongs to the domain of the function. For example, the following is part of the proof obligation generated by Z/EVES for the *Vote* operation:

VoteOk

$\Delta Votes;\ \Xi Endorsements$
$e? : RSCH$
$C? : \mathbb{F}\, RSCH$

$bVote \leq today \leq fVote$	$\mathsf{Pre}-1$
$e? \in CanVote$	$\mathsf{Pre}-2$
$e? \notin voted$	$\mathsf{Pre}-3$
$\#C? \leq 3$	$\mathsf{Pre}-4$
$C? \subseteq firmCand$	$\mathsf{Pre}-5$
$\{myKA\ e?\} = myKA(\!\vert C?\vert\!)$	$\mathsf{Pre}-6$
$\#C? = \#(myGR(\!\vert C?\vert\!))$	$\mathsf{Pre}-7$
$voted' = voted \cup \{e?\}$	$\mathsf{Pos}-1$
$votes' = votes \frown \langle C? \rangle$	$\mathsf{Pos}-2$
$firmCand' = firmCand$	

Fig. 2. *VoteOk* is the main schema of the *Vote* operation.

theorem axiom Vote$domainCheck
$\Delta Votes \land \Xi Endorsements \land \Xi Time$
$\land\ e? \in RSCH \land C? \in \mathbb{F}\, RSCH$
$\Rightarrow (iVote \leq today \land \cdots \land \{myKA\ e?\} = myKA(\!\vert C?\vert\!)$
$\quad \Rightarrow C? \in \mathrm{dom}\,\# \land myGR(\!\vert C?\vert\!) \in \mathrm{dom}\,\#)$
. . . . . .

where the difficult part is to prove that $myGR(\!\vert C?\vert\!) \in \mathrm{dom}\,\#$ because it requires to prove that the relational image of a finite set through a function is a finite set. We proved this and other theorems involving the cardinality operator with the help of the extension to the Z mathematical toolkit (ZMT) proposed in [15].

Then we proved that all operations preserve the state invariants shown in Fig. 1. We did this by proving a theorem like VotePreservesInvariants shown above, for each operation of the model. However, proving such a theorem can be cumbersome because some parts of the proof can be quite difficult and there are many cases to consider since each operation includes many schemas and there are many invariants. But, on the other hand, many of these cases are trivial to prove. Hence, we analyzed which invariants would be, in principle, non trivial to prove for a given operation and thus we defined a theorem for each of these cases. For example we have:

theorem EndorseOkPI5
$\mathrm{dom}\ cand = \mathrm{dom}\ endors$
$\land\ (\forall\, i, j : \mathrm{dom}\ endors \mid i \neq j \bullet endors\, i \cap endors\, j = \emptyset)$
$\land\ EndorseOk \Rightarrow \forall\, i, j : \mathrm{dom}\ endors' \mid i \neq j \bullet endors'\, i \cap endors'\, j = \emptyset$

as an intermediate theorem for the operation representing a voter endorsing a preliminary candidate. Note that in this case we prove that only the schema cor-

responding to the successful case (*EndorseOk*) preserves only the fifth invariant (see the fifth predicate in schema *Invariants* at Fig. 1).

Finally, in this project we needed to prove five theorems involving mathematical results. For example, we proved that the relational image of a finite set through a partial function is a finite set:

theorem grule finiteRelimgIsFinite $[X, Y]$
$\forall f : X \pfun Y;\ A : \mathbb{F}\, X \bullet f(\!| A |\!) \in \mathbb{F}\, Y$

All the proofs performed during this step gave us a reasonable confidence that the specification is a faithful formalization of the requirements document, hence the specification can be used as the guide for verification.

4 Specification-Guided Code Inspection

Having confidence on the correctness of the specification is of no help to users if it is not used as a means to gain confidence on the correctness of the implementation. Our approach to gain confidence on the correctness of the implementation was based on: a) inspecting the code to see if it refines the specification; and b) run some test cases derived from the specification. In this section we explain how we used the specification to inspect the source code of the application.

CONICET decided to program the web voting system as a Grails application. Grails [32] is a web application framework for the Java Virtual Machine which in turn takes advantage of the Groovy programming language [33]. That is, the application is an object-oriented program written in a combination of high-level programming languages (Java and Groovy) and complex frameworks (Grails). We cannot show nor make public the full application for confidentiality issues, but we will show excerpts of it.

We used the specification as the guide to conduct the inspection of the source code. Therefore, the first step was to identify the implementation data structures and variables that refine[4] the state variables declared in schemas *Endorsements* and *Votes* and the input variables declared in the operation schemas like *Vote* (Sect. 2). This is documented in a table similar to Table 1. Note that building such a table might not be trivial because the specification was written from the requirements document and not from the code. For example, the requirements document, and thus the specification, does not mention data encryption at all, but it is used in the implementation. As another example, the implementation of variable *firmCand* is a table in a database called `BallotPaper` which holds data about the candidates that voters may vote and a variable in memory, `papers`, which, at some point, is assigned with the contents of that table through a Grails' mechanism.

Once we had clear how state variables were implemented we identified in the source code all the major operations of the specification. This involves to find

[4] In this paper the word "refine" is not used with the formal meaning it has in Z. The intention of the code inspection is to informally evaluate whether the program is a correct implementation of the specification.

Table 1. Implementation variables associated to specification variables.

Specification	Implementation	Comments
firmCand	`BallotPaper`, `papers`	`BallotPaper` is a table in the database. `papers` is a variable in memory.
voted	`Elector`	Is a table in the database. If `Date` is non-empty, elector has voted.
votes	`EncryptedVote`	Is a table in the database. Each record is an encrypted instance of `Vote`.
C?	`Vote`	Includes also the election and GR of candidates.

the relation between the signature of operations at the specification level and subroutines in the implementation, and check whether they match or not.

The last step during the code inspection was to check whether all the pre and postconditions recorded in operation schemas, like *Vote*, are correctly implemented. Due to the differences in the structure of the implementation and the specification and those introduced by the (implicit or *de-facto*) refinement made by programmers, this step is perhaps the most difficult to do although the more likely to uncover errors. Firstly, we labeled each pre and post-condition in the specification as can be seen in schema *VoteOk* (Fig. 2). Secondly, we used the information gathered in the two previous steps to focus the inspection on specific implementation units. For example, in order to inspect the implementation of operation *Vote* we read code in `VoteController.groovy` and `VoteService.groovy`, and we looked for the refinement of variables *voted*, *firmCand*, *votes*, etc.

```
class VoteController {
  VoteService voteService

  def beforeInterceptor = {
    def userPrincipal = (AttributePrincipal)
      RCH.currentRequestAttributes().request.userPrincipal
    def elector =
      Elector.findByUserId(userPrincipal.attributes.usrnum)

//Begin Vote::Pre-3, Vote::Pos-3
  if (elector.voteDate) {
    render 'You already voted.'
    return false
  }
//End Vote::Pre-3, Vote::Pos-3
...
```

Fig. 3. Class `VoteController` annotated with links to the specification.

Every time we found the implementation of a pre or post-condition we annotated the program as shown in Fig. 3, i.e. by using the labels introduced in operation schemas. See the comments before and after the conditional structure. $\mathsf{Pre}-3$ is one of the preconditions of schema *VoteOk* ($e? \notin voted$); $\mathsf{Pos}-3$ is a postcondition present in schema *AlreadyVoted*. Observe how the precondition was implemented: instead of having a table or file storing all the persons who have issued their votes so far, programmers decided to augment the electoral roll with an extra column that if empty means that the corresponding person has not voted and otherwise it stores the date when the person issued his or her vote. As can be seen, pre and postconditions may be annotated together because the "else" branch of conditional sentences is not always present.

Clearly, the mere presence of a sentence implementing a given condition in the specification does not guarantee the correctness of the implementation. Correctness depends also on the sentences before and after the one that has been annotated. However, once the implementation has been inspected, evaluated as correct and annotated in this way, a convenient IDE can assist the development team during maintenance because the tool can bring specifics predicates of the specification into attention, can show all the pieces of code implementing a given condition, etc.

5 Generating Test Cases from the Specification

The Z specification was also used to generate test cases to exercise the implementation. Testing the implementation is important even after code inspection because many third-party components with which the application interacts may fail. This is particularly important for the application being considered because it is implemented over and interacts with very complex components like the JVM, Groovy, Grails, MySQL, etc.

Test cases were generated by following a model-based testing method known as Test Template Framework (TTF) and by using Fastest, a tool that semi-automates the TTF. Given that the TTF and Fastest have been extensively described [10], here we will show, by means of an example, how we applied them to generate test cases. They are used for functional unit testing.

Consider schema *Vote* (Sect. 2) and its input space—i.e., all its input and state variables. First, we partition the input space of *Vote* by applying Disjunctive Normal Form (DNF) in which case Fastest generates nine test conditions, including the three shown in Fig. 4. Note that these schemas include only input and (unprimed) state variables. DNF guarantees that the main situations described in *Vote* are going to be tested. For instance, a user successfully issuing the vote ($Vote_1^{DNF}$); a user trying to vote after the election ($Vote_3^{DNF}$); and a user trying to vote more than once ($Vote_5^{DNF}$). Any of these test conditions can be further partitioned. For example, we can partition $Vote_1^{DNF}$ by applying a standard partition to the operator $\cup$ in $voted' = voted \cup \{e?\}$, thus yielding the following test conditions among others:

$Vote_1^{DNF}$
$Vote^{IS}$

$bVote \leq today \leq fVote$
$e? \in CanVote$
$e? \notin voted$
$0 \leq \#C?$
$\#C? \leq 3$
$C? \subseteq firmCand$
$\{myKA\, e?\} = myKA(\!|\, C? \,|\!)$
$\#C? = \#(myGR(\!|\, C? \,|\!))$

$Vote_3^{DNF}$
$Vote^{IS}$

$fVote < today$

$Vote_5^{DNF}$
$Vote^{IS}$

$e? \in voted$

Fig. 4. Some test conditions. $Vote^{IS}$ represents $Vote$'s input space.

$Vote_1^{SP}$
$Vote_1^{DNF}$

$voted = \emptyset$
$\{e?\} = \emptyset$

$Vote_2^{SP}$
$Vote_1^{DNF}$

$voted = \emptyset$
$\{e?\} \neq \emptyset$

$Vote_4^{SP}$
$Vote_1^{DNF}$

$voted \neq \emptyset$
$\{e?\} \neq \emptyset$
$voted \cap \{e?\} = \emptyset$

$Vote_7^{SP}$
$Vote_1^{DNF}$

$voted \neq \emptyset$
$\{e?\} \neq \emptyset$
$\{e?\} = voted$

As can be seen, some of the conditions are unsatisfiable ($Vote_1^{SP}$, for instance) but Fastest implements an algorithm that can eliminate many of them [10]. Note that these test conditions include the conditions of $Vote_1^{DNF}$ because this schema is included in the others. Therefore, for example, a test case derived from $Vote_2^{SP}$ will exercise the system when a user is allowed to vote, issues a valid vote and is the first person to vote; while $Vote_4^{SP}$ will do the same but there should be another user who has already voted.

Once engineers are done with partitioning they can generate test cases from the last test conditions. Below we show two typical test cases generated by Fastest ($rsch0$ is a constant of type $RSCH$ identifying a particular researcher).

$Vote_2^{TC}$
$Vote_2^{SP}$

$today = 5$
$e? = rsch5$
$C? = \{rsch0\}$
$firmCand = \{rsch0\}$
$voted = \emptyset$
$votes = \langle\rangle$

$Vote_4^{TC}$
$Vote_4^{SP}$

$today = 5$
$e? = rsch5$
$C? = \{rsch0\}$
$firmCand = \{rsch0\}$
$voted = \{rsch0\}$
$votes = \langle\{rsch0\}\rangle$

These test cases, however, are Z terms that cannot be executed by the application. Hence, we gave developers precise instructions on how to write JUnit [1] test cases from the Z test cases, and we translated some of them as examples. For writing JUnit test cases we used some of the information gathered during the code inspection activity. For example, we used the data structures we identified as the refinement of the state and input variables. That is, each activity provides useful data for the other activities.

6 Discussion

The Z specification presented here is about 450 lines of Z code in a 20 pages document. The implementation has approximately 2,575 lines of Grails and Java code. We proved 31 theorems, 7 of which were discharged automatically. 9 of the theorems are domain checks automatically generated by Z/EVES; 5 are theorems about mathematical properties; 6 are the main theorems (i.e. those proving that each operation preserves all the invariants); and 11 are auxiliary theorems about the specification. The proof scripts total 2,045 lines of commands. We generated only 68 test cases mainly because we were running out of time and because developers will not have time to run more. However, we estimate that Fastest could generate around 200 test cases. The test cases that were generated cover the main functional alternatives of all the operations.

The specification was written in 20 man-hours and its verification took around 80 man-hours. Code inspection was performed in 32 man-hours although it could have been less provided we have had a deeper knowledge of the implementation technologies. Generating the test cases took less than 8 man-hours mainly because the specification contains many axiomatic descriptions whose values are underspecified and must be fully specified before test case generation. Then, our work totaled around 140 man-hours. We do not have figures about test case refinement and execution because it was performed by developers.

During the verification of the specification four minor errors were found. Three of them concerned with declaring some variables as $\mathbb{P}$ instead of $\mathbb{F}$. The remaining one was a missing precondition in one operation. The code inspection revealed that only two operations of the model were implemented (*Vote* and *Recount*). That is, all the information regarding preliminary and firm candidates and the electoral roll, is loaded manually from different sources. We are not sure

whether top management was aware of this fact. In the implemented operations code inspection revealed no errors. Performing a code inspection before engaging in testing can make testing very cost-effective since many errors are discovered during the inspection. Furthermore, the code inspection yields a documented project (specification and implementation) making it easier to introduce changes and bug-fixes. As far as we have been told, test cases were run and all the errors found where corrected.

We know that there have been verification efforts producing verified software that involve larger specifications and implementations than the one presented here—see Section 7. However, our project presents the following crucial differences with many, if not all, of them:

1. Usually, these projects are performed entirely, including the programming stage, by experts on formal verification (many of them are either PhD students or people already holding a PhD); or
2. The verification stage does not cover a full functional assessment. In this case, it usually encompass the verification of general properties such as code-level safety, noninterference, etc. that in many cases can be fully automatized.

Furthermore, in many cases, the implementation is developed once the specification has been written and proved correct. Many formal techniques have been thought to be applied if a correct specification exists. In many of these projects the creators of the techniques being applied are those who run them.

In contrast, the project presented here involves the verification of a program that was developed by average programmers before the verification activities were even thought. This means, for example, that we were not allowed to chose the programming language. In other words, verification was an afterthought and the implementation was not done by researchers, as it is in almost all the software produced nowadays. On the other hand, the verification comprised a full functional verification which involved specific properties of this voting system. Even the verification of general properties (such as code-level safety) was not possible in this project due to the implementation technology chosen by developers. In summary, we think that the project presented here is closer to the way the software industry works, making it more appealing to evaluate the applicability of formal methods. It remains as a challenge to verify a lager project with this methodology.

7 Related Work

The Z notation, and its extensions, is being used for formal specification since the early eighties. It has been used to specify a wide range of systems; we will mention just a few of the latest specifications to give an idea of the kind of requirements formalized with Z. Perhaps one of the most praised Z specifications in recent years is the Tokeneer abstract specification [4]. This project was an experiment performed by NSA to prove that formal methods are cost-effective in real-world software. Altran Praxis was finally hired for the job. They wrote a Z specification

that was later verified to be correct by proving some security properties. Also a low level design was written in Z and the SPARK code was formally verified and reviewed. The net result was that only two errors were discovered after delivery although more recent studies show that more errors exist [27]. Altran Praxis has used these technologies in many projects of different application domains.

Cristiá and others have used Z for modeling aerospace software such as satellite communication protocols, part of a launching vehicle control software and the ECSS-E-70-41A aerospace standard [12, 11]. In all these projects the TTF and Fastest were applied to generate test cases.

Frydman and her colleagues have combined Z with DEVS [41] for the validation of discrete event systems via simulation and formal methods [34, 36]. Object-Z has been used in the formalization of the Web Service Modeling Ontology for the Semantic Web [37] and also Z and Z/EVES were applied in the same domain [24]. The main purpose of these works was to provide a precise and unambiguous specification for concepts that have traditionally been informal.

Security systems has also been the focus of Z specifications. For example, in [19] the authors combined Z and CSP [29] to provide a formal specification for the Audited Credential Delegation architecture which would help virtual organizations in managing the identities of their users. Security is often a critical aspect of some systems, but there are systems that are critical in their own. Gomes and Oliveira [18] have written a Z specification for a cardiac pacing system which is one of the challenges proposed as part of the Verification Grand Challenge. This specification comprises 4,000 lines of Z. As a last example, we can mention the specification of the safety properties of a railway interlocking system [40], which is one of the traditional targets of formal specification due to the potential damages a failure may cause.

Z/EVES has been the proof assistant used in many projects involving Z specifications. Khan et al. [24] and Zafar and his colleagues [40] use Z/EVES to discharge the proof obligations automatically generated by the tool itself. However, some have used Z/EVES to prove properties of the specification as a means to gain confidence on its correctness. For example, in [35] the authors used this tool to prove 40 theorems about feature modeling. Amalio and others used Z/EVES to prove properties of FTL, a formal language that allows the description of formal templates written in any formal language, in particular templates for the Z notation [3] and for the UML modeling notation [2]. Dong and Wang [13] explore the benefits of using Z/EVES to detect inconsistencies in semantic web services, although it seems that they proved a small amount of properties. Yuan and others [39] used Z/EVES to prove some security properties (separation of duty, in this case) of a state-based role-based access control (RBAC) model. Freitas and Woodcock have extensively used Z/EVES for proving properties of complex systems such as the Mondex Electronic Purse [38, 17] and a POSIX file store [16], contributing to the verified software repository. Cristiá and other have used Z/EVES to prove that so-called pruning rules are sound and so they can be used to eliminate inconsistent test conditions from testing trees [10].

Program annotation has a long tradition in the formal methods community and in other fields of programming; we will review some representative works. Cataño and Huisman [8] annotate an application with ESC/Java [9] in such a way that a formal functional specification is provided. The authors say that application developers might be more inclined to write formal specifications if specifications are written in a language closer to the programming language being used. While this might indeed be true, a disadvantage of this approach is that the specification becomes less abstract. This work reports a serious impact on the quality and documentation of the project. Developers at Altran Praxis annotate their SPARK programs with data and information flow clauses that are later analyzed by the SPARK Examiner [25]. They also annotate programs with pre and postconditions to perform a functional verification. In this case the SPARK Examiner generates proof obligations that, in general, cannot be discharged automatically. This work is also interesting because some of the SPARK annotations are derived from the Z specification by following some naming conventions and by running simple type translations. Note, however, that the broader context of the project is quite different: in the Altran Praxis case they developed the specification and the implementation allowing them to select the latter; in our case, we could not select the implementation because it was already given—for instance, we could not decide to implement the voting system in SPARK. There are many other works investigating different aspects of program annotation but they are not closely related with the ideas presented here [5, 23, 26, 14]

Some properties of larger programs than the one presented here were automatically proved correct by means of many static analysis techniques. For example, Berdine et al. describe a tool (Slayer) that can automatically prove memory safety of industrial systems such as Windows device drivers [6]. Another advanced tool that was applied to large, safety-critical, embedded, real-time software is presented by Blanchet and his colleagues [7]. They acknowledge that their tool works for a restricted class of programs and properties. VeriFast and Frama-C are another two static analysis tools that have been applied to large programs [28, 20]. Although all these tools represent remarkable achievements in their field, the proven properties are not functional or do not fully cover a functional verification of the implementation.

We could not find works presenting the combined application of all the techniques shown in this paper in the same project, that is: formal specification, formal verification of the specification, program annotation with respect to the specification as the basis for code inspection, and model-based testing as a complementary verification activity.

8 Conclusions

During the verification of the voting system reported in this paper we applied four techniques ranging from formal to informal ones. Given the resources and time available we were able to transmit to senior management a reasonable level

of confidence on the correctness of the application. The first election was carried out without any noticeable failure.

In our opinion, the most valuable contribution of this report is that these techniques were applied to an industrial project under very realistic conditions. That is, a project where average developers implemented a program whose full functional verification was scheduled once it was finished. When this is the case, verification has a very low budget and tight schedule. In this context, these techniques proved to be effective and efficient. Moreover, the separation between a standard team of developers and a group of researchers in charge of the verification may be a good strategy in many projects. However, this setting posses some challenges to formal methods. The main reason is that the verification team has practically no influence on the implementation technologies. In turn, this implies that many formal techniques and tools cannot be applied or they must be adapted or lightened.

We believe that the combination between a formal specification, a code inspection guided by the specification and model-based testing (as was done in this project) might be the basis of a verification methodology for mission critical applications whose verification is requested once the implementation is finished.

References

1. JUnit.org Resources for Test Driven Development, `http://junit.org/`, last access: November 2011
2. Amálio, N., Stepney, S., Polack, F.: Formal proof from uml models. In: Davies, J., Schulte, W., Barnett, M. (eds.) ICFEM. Lecture Notes in Computer Science, vol. 3308, pp. 418–433. Springer (2004)
3. Amálio, N., Stepney, S., Polack, F.: A formal template language enabling metaproof. In: Misra, J., Nipkow, T., Sekerinski, E. (eds.) FM. Lecture Notes in Computer Science, vol. 4085, pp. 252–267. Springer (2006)
4. Barnes, J., Chapman, R., Johnson, R., Widmaier, J., Cooper, D., Everett, B.: Engineering the Tokeneer enclave protection software. In: Proceedings of the IEEE International Symposium on Secure Software Engineering. IEEE (2006)
5. Bartetzko, D., Fischer, C., Möller, M., Wehrheim, H.: Jass - java with assertions. Electr. Notes Theor. Comput. Sci. 55(2), 103–117 (2001)
6. Berdine, J., Cook, B., Ishtiaq, S.: Slayer: Memory safety for systems-level code. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV. Lecture Notes in Computer Science, vol. 6806, pp. 178–183. Springer (2011)
7. Blanchet, B., Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A., Monniaux, D., Rival, X.: The essence of computation. chap. Design and implementation of a special-purpose static program analyzer for safety-critical real-time embedded software, pp. 85–108. Springer-Verlag New York, Inc., New York, NY, USA (2002), `http://dl.acm.org/citation.cfm?id=860256.860262`
8. Cataño, N., Huisman, M.: Formal specification and static checking of gemplus' electronic purse using ESC/Java. In: Proceedings of the International Symposium of Formal Methods Europe on Formal Methods - Getting IT Right. pp. 272–289. FME '02, Springer-Verlag, London, UK, UK (2002), `http://dl.acm.org/citation.cfm?id=647541.730158`

9. Compaq Systems Research Center: Extended static checking for Java, http://web.archive.org/web/20051208055447/http://research.compaq.com/SRC/esc/, last access: February 2013
10. Cristiá, M., Albertengo, P., Frydman, C., Plüss, B., Monetti, P.R.: Tool support for the test template framework. Software Testing, Verification and Reliability 24(1), 3–37 (2014), http://dx.doi.org/10.1002/stvr.1477
11. Cristiá, M., Albertengo, P., Frydman, C.S., Plüss, B., Monetti, P.R.: Applying the Test Template Framework to aerospace software. In: Rash, J.L., Rouff, C. (eds.) SEW. pp. 128–137. IEEE Computer Society (2011)
12. Cristiá, M., Santiago, V., Vijaykumar, N.: On comparing and complementing two MBT approaches. In: Test Workshop (LATW), 2010 11th Latin American. pp. 1–6 (2010)
13. Dong, J.S., Sun, J., Wang, H.H.: Z approach to semantic web. In: George, C., Miao, H. (eds.) ICFEM. Lecture Notes in Computer Science, vol. 2495, pp. 156–167. Springer (2002)
14. Frade, M.J., Pinto, J.S.: Verification conditions for source-level imperative programs. Computer Science Review 5(3), 252 – 277 (2011), http://www.sciencedirect.com/science/article/pii/S1574013711000037
15. Freitas, L.: Z/Eves extended Z toolkit. Tech. rep., University of York (2004)
16. Freitas, L., Fu, Z., Woodcock, J.: Posix file store in z/eves: an experiment in the verified software repository. Engineering of Complex Computer Systems, IEEE International Conference on 0, 3–14 (2007)
17. Freitas, L., Woodcock, J.: Mechanising mondex with z/eves. Formal Aspects of Computing 20, 117–139 (2008), http://dx.doi.org/10.1007/s00165-007-0059-y
18. Gomes, A.O., Oliveira, M.V.: Formal specification of a cardiac pacing system. In: Proceedings of the 2nd World Congress on Formal Methods. pp. 692–707. FM '09, Springer-Verlag, Berlin, Heidelberg (2009), http://dx.doi.org/10.1007/978-3-642-05089-3_44
19. Haidar, A.N., Coveney, P.V., Abdallah, A.E., Ryan, P.Y.A., Beckles, B., Brooke, J.M., Jones, M.A.S.: Formal modelling of a usable identity management solution for virtual organisations. In: Bryans, J., Fitzgerald, J.S. (eds.) FAVO. EPTCS, vol. 16, pp. 41–50 (2009)
20. Hartig, K., Gerlach, J., Soto, J., Busse, J.: Formal specification and automated verification of safety-critical requirements of a railway vehicle with frama-c/jessie. In: Schnieder, E., Tarnai, G. (eds.) FORMS/FORMAT. pp. 145–153. Springer (2010)
21. ISO: Information Technology – Z Formal Specification Notation – Syntax, Type System and Semantics. Tech. Rep. ISO/IEC 13568, International Organization for Standardization (2002)
22. Jackson, M.: Software requirements & specifications: a lexicon of practice, principles and prejudices. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA (1995)
23. Jacobs, B.: Weakest pre-condition reasoning for java programs with jml annotations. J. Log. Algebr. Program. 58(1-2), 61–88 (2004)
24. Khan, S.A., Hashmi, A.A., Alhumaidan, F., Zafar, N.A.: Semantic web specification using Z-notation. Life Science Journal 9(4) (2012)
25. King, S., Hammond, J., Chapman, R., Pryor, A.: Is proof more cost-effective than testing? IEEE Trans. Software Eng. 26(8), 675–686 (2000)

26. Marché, C., Paulin-Mohring, C., Urbain, X.: The krakatoa tool for certificationof java/javacard programs annotated in jml. J. Log. Algebr. Program. 58(1-2), 89–106 (2004)
27. Moy, Y., Wallenburg, A.: Tokeneer: Beyond formal program verification. In: Proc. 5th Int. Congress on Embedded Real Time Software and Systems (ERTS'10). Toulouse, France (May 2010)
28. Philippaerts, P., Mühlberg, J.T., Penninckx, W., Smans, J., Jacobs, B., Piessens, F.: Software verification with VeriFast: Industrial case studies. Science of Computer Programming (0), – (2013), http://www.sciencedirect.com/science/article/pii/S0167642313000191
29. Roscoe, A.W.: The Theory and Practice of Concurrency. Prentice Hall PTR, Upper Saddle River, NJ, USA (1997)
30. Saaltink, M.: The Z/EVES 2.0 User's Guide. Ora Canada (1999)
31. Spivey, J.M.: The Z notation: a reference manual. Prentice Hall International (UK) Ltd., Hertfordshire, UK, UK (1992)
32. SpringSource: Grails – The search is over, http://grails.org/, last access: February 2013
33. SpringSource: Groovy – A dynamic language for the Java platform, http://groovy.codehaus.org/, last access: February 2013
34. Sqali, M., Trojet, W., Torres, L., Frydman, C.: Combining interaction and state based modelling to validate system specification via simulation and formal methods. In: Winter Simulation Conference (WSC 2009) Poster Session. Austin, Texas (december 2009)
35. Sun, J., Zhang, H., Wang, H.: Formal Semantics and Verification for Feature Modeling. In: ICECCS '05: Proceedings of the 10th IEEE International Conference on Engineering of Complex Computer Systems. pp. 303–312. IEEE Computer Society, Washington, DC, USA (June 2005)
36. Trojet, W., Sqali, M., Frydman, C., Torres, L., el-amine Hamri, M.: Validating the global behaviour of a system described with scenarios using GDEVS and Z. In: 21th European Modeling and Simulation Symposium (EMSSÓ9). Tenerife - Canary Islands, Spain (september 2009)
37. Wang, H.H., Gibbins, N., Payne, T.R., Redavid, D.: A formal model of the semantic web service ontology (WSMO). Information Systems 37(1), 33 – 60 (2012), http://www.sciencedirect.com/science/article/pii/S0306437911001049
38. Woodcock, J., Freitas, L.: Z/eves and the mondex electronic purse. In: Barkaoui, K., Cavalcanti, A., Cerone, A. (eds.) Theoretical Aspects of Computing - ICTAC 2006, Lecture Notes in Computer Science, vol. 4281, pp. 15–34. Springer Berlin Heidelberg (2006), http://dx.doi.org/10.1007/11921240_2
39. Yuan, C., He, Y., He, J., Zhou, Z.: A verifiable formal specification for rbac model with constraints of separation of duty. In: Lipmaa, H., Yung, M., Lin, D. (eds.) Information Security and Cryptology, Lecture Notes in Computer Science, vol. 4318, pp. 196–210. Springer Berlin Heidelberg (2006), http://dx.doi.org/10.1007/11937807_16
40. Zafar, N.A., Khan, S.A., Araki, K.: Towards the safety properties of moving block railway interlocking system. Int. J. Innovative Comput., Info & Control (2012)
41. Zeigler, B.P., Kim, T.G., Praehofer, H.: Theory of Modeling and Simulation. Academic Press, Inc., Orlando, FL, USA (2000)