Formal and Semi-formal Verification of a Web Voting System

No Author Given

No Institute Given

Abstract.
Purpose: CONICET is the most important research institution in Argentina. It depends directly from Argentina's President but 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 process. In 2012 and 2014 CONICET conducted two web election with no complains from candidates and voters. Before moving the system into production, CONICET asked the authors to conduct a functional and security assessment of it.In this paper we present the verification process conducted to assess the functional correctness of the voting system.
Design/methodology/approach: This process is the result of integrating formal, semi-formal and informal verification activities from formal proof to code inspection and model-based testing.
Findings: 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.
Research limitations/implications: A formal specification of the requirements must be developed.
Practical implications: N/A
Social Implications: N/A
Originality/value: Formal methods and semi-formal activities are seldom applied to Web applications.

1 Introduction

The Consejo Nacional de Investigaciones Científicas y Técnicas (CONICET[1]) is the most important research institution in Argentina. Its president is nominated by Argentina's President but 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 by mail (i.e. those allowed to vote had to send their votes by mail and, thus, the recount process was done 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

[1] In English: National Scientific and Technical Research Council

system[2]. According to CONICET's authorities its web-based voting system is the first to be developed and used in Argentina.

As part of this important change, the Board of Directors issued a set of regulations as the legal support for the new form of the election process. These regulations set all the rules that voters and candidates have to follow in order to vote or be voted. They take the form of a document written in natural language (Spanish) organized as a list of clauses.

The Board of Directors also 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. The System Department contacted the authors to perform the evaluation. One of the authors 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. Note that 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 (Sect. 2). Secondly, we formally proved that the specification verifies some properties as a way to have some confidence that the model is correct (Sect. 3). The third step consisted in inspecting the source code to find each pre and postcondition of the model, and annotating the source code with links to the corresponding predicates of the formal specification (Sect. 4). In the last step we applied Fastest, a model-based testing (MBT) tool, to generate test cases (Sect. 5). In this way, the formal specification written in the first step was used in all the latter activities. As can be seen, the verification process combined formal, semi-formal and informal techniques making it possible to finish the task within the schedule. In the rest of the paper we further discusses these and other issues (Sect. 6) and survey some related papers (Sect. 7); our conclusions are in Sect. 8.

It is very important to remark that this project presents two important differences with respect to other verification efforts. Firstly, in most other reports on the production of verified software the development team can write a formal specification of the system and then develop the implementation, they like the most, using many different formal techniques—formal refinement, program annotation combined with static analysis, etc. In this project, however, the implementation was already developed when its verification was requested—i.e. verification was an afterthought as is always the case in the vast majority of industrial projects. Therefore, we were not allowed to generate the implementation we liked—mainly because of the available time—; we had to inform about the correctness of the available implementation. Secondly, the implementation was

[2] CONICET's Systems Department is not a research unit.

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. No voter and candidate complained about the election process or its outcome. The same happen after the 2014 election.

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`. This paper is also an extended version of a conference paper [1]. It assumes the reader has a basic knowledge of set theory and predicate logic.

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 specification as the fundamental cornerstone of the verification process. We knew in advance that we could use it for many different verification activities. The Z notation [2, 3] was the chosen language because: a) we are fluent in it and some of its tools; b) Z is very good for this kind of requirements; and c) our research and tools on MBT [4–6] would be of great help (and conversely, this would become a case study for our investigations).

Although the election process of CONICET shares many rules with that of any democratic nation, it has some peculiarities that distinguish it from other election systems. Just to mention some of the differences consider the following:

- CONICET divides the country in geographical regions (GR).
- CONICET divides its researchers in so-called Knowledge Areas (KA) (for instance, Exact and Natural Sciences, Social Sciences and Humanities, etc.).
- Each voter belongs to one and only one GR and KA.
- Each candidate candidates her or himself for one and only one GR and KA.
- Each voter can vote up to three candidates of his KA but of different GR.
- Retired researchers may be allowed to vote.
- Not all active or retired researchers can vote or be candidates, there are many conditions they must verify for either one.
- Before a voter can be designated as a firm candidate, he or she has to be endorsed by at least 20 researchers allowed to vote.

The Z specification resulting from the requirements document is a rather standard Z model[3]. We added to the Z model a form of Jackson's designations [7] 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 semi-formal description (designation):

x 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 a natural language sentence over the vocabulary of the requirements while the right hand side is a formal term of the model. Since this is not a technical document we will not give all the designations corresponding to the portions of the specification included here; we prefer to be more informal in this regard.

2.1 A Brief Introduction to Z

Z specifications take the form of state machines. That is, there is a state space and a set of state transitions defined over the state space. The state space is given by declaring a set of state variables, each ranging over a particular set of values. Then, the first step in a Z specification is to declare the sets or types of values for the state variables. These sets, as the rest of the elements of the specification, are taken from the requirements. Hence, we have defined the following basic sets:

- The set of knowledge areas, named KA
- The set of CONICET's researchers, named $RSCH$
- The set of geographical regions in which the country is divided to organize the election, named GR

Now, we can give the main components of the state of the system. We factored out the state of the system into two schemata. *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

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

[3] In the project we used both Spivey's Z for Z/EVES and standard Z [3] for Fastest; in the paper we use only the latter.

Think of these boxes, called schemas, as sets of elements. In Z: $RSCH \pfun KA$ denotes the set of all partial functions from $RSCH$ to KA; $\mathbb{F}\, RSCH$ is the set of all the finite sets whose elements belong to $RSCH$; $\mathbb{P}\, RSCH$ is the set of all the sets (finite or not) whose elements belong to $RSCH$; and $\mathrm{seq}(\mathbb{F}\, RSCH)$ is the set of finite sequences (lists) whose elements belong to $\mathbb{F}\, RSCH$. In Z, partial functions are sets of ordered pairs. So, for instance, a partial function can be expressed as follows: $\{a \mapsto 1, b \mapsto 2\}$, where $a \mapsto 1$ is a synonym for the more standard notation $(a, 1)$.

The meaning of state variables is as follows:

- if $x \mapsto a \in cand$ then x is a candidate willing to be endorsed for KA a;
- if $x \in endors(c)$ then voter x endorsed candidate c;
- if $x \in firmCand$ then x is a firm candidate (i.e. he or she has been endorsed according to CONICET rules);
- if $x \in voted$ then x has already issued his or her vote; and
- $votes$ is the list of votes issued so far.

We used $\mathbb{P}$ instead of $\mathbb{F}$ for a variable whenever it was not necessary a cardinality condition (for example, there is no cardinality condition over $firmCand$). Note the particular type for $votes$: is not just a list of researchers but a list of finite sets of researchers since voters can vote for up to three candidates. It would has been better to declare $votes$ as $\mathrm{bag}(\mathbb{F}\, RSCH)$ but Fastest (the MBT tool used for testing) does not currently supports bags.

The full state space of the election system is defined by schema ES (for election system) as follows:

$$ES == Endorsements \land Votes \land Today$$

where schema $Today$ declares a single variable representing the current date.

Once we have the state space of the system, i.e. schema ES, we can specify the state transitions. Due to space restrictions we will show only one of them—the rest is available in the technical report. State transitions are called operations, in Z. Each operation is given by a schema—as can be seen, in Z schemas are used for many purposes. Below we give the schema for the the operation describing the conditions for a voter to issue his or her vote. The schema is called $Vote$ and is given as the disjunction of eight schemas:

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

Think of such an expression as a nested conditional sentence of a programming language: $\mathsf{if} \ldots \mathsf{then} \ldots \mathsf{else\ if} \ldots .$ $VoteOk$ formalizes the situation when a

voter successfully issues his or her vote; all the other schemas describe possible errors, and so the vote is not recorded.

In *VoteOk*, shown below, $e?$ is the researcher issuing the vote and $C?$ is his or her vote.

VoteOk

$\Delta Votes$
$\Xi Endorsements;\ \Xi Today$
$e? : RSCH$
$C? : \mathbb{F}\ RSCH$
$rsp! : MSG$

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

For now do not pay too much attention to the expressions $\Delta Votes$ and $\Xi Endorsements;\ \Xi Today$, we will explain them later. As we have said, Z operations are state transitions. Then, they link two states of the state space: the start or before state and the after state. Think of states as the initial and final contents of a table in a database. For example, if a method adds a record to a table it has changed the state of the table because the initial table is different from the final one. Z operations can also depend on input and output variables. Input variables are decorated with a question mark. So, for instance, ? and $C?$ are input variables. Think of input variables as the parameters waited by a method. Output variables are decorated with a bang symbol. So, $rsp!$ is an output variable. Think of output variables as the values returned by a method.

Any Z operation is described by giving its pre- and post-conditions. Perconditions are predicates depending only on before-state and input variables. Before-state variables are the variables declared in schemas *Votes* and *Endorsements*. Post-conditions are predicates that can depend on any kind of variable. Usually post-conditions depend on before-state and after-state variables. After-state variables are the before-state variables but decorated with a prime symbol. So, for example, $voted'$ is an after-state variable. Before-state and after-state variables are made accessible to *Vote* by the expressions $\Delta Votes$ and $\Xi Endorsements$. Indeed, $\Delta Votes$ means "*Vote* may change the value of some of the state variables declared in *Votes*"; while $\Xi Endorsements$ means "*Vote* cannot change the value of the state variables declared in *Endorsements*".

Now, let us take a closer look to *Vote*. Recall that the vote, $C?$, may contain up to three candidates so it is defined as a finite set of researchers. Then we include the pre-condition $\#C? \leq 3$ to enforce that at most three candidates can be voted. $bVote$ and $fVote$ are two "dates" representing the initial and final date of the voting period. *today* is a state variable declared in schema *Today*, which represents the current date. So the precondition $bVote \leq today \leq fVote$ says that a vote can be issued only of the current date is between the initial and final dates of the voting period. *CanVote* is the set of CONICET's researchers that are allowed to vote. It is defined as follows:

$$\begin{array}{l} CanVote == \\ \quad \{i : RSCH \mid \\ \qquad myKA\, i \in eKA \land (i \in ret \Rightarrow i \in hiredRet) \land i \in inIntranet\} \end{array}$$

where *myKA* is a function that yields the *KA* of each researcher; *eKA* is the set of *KA*'s for which the current election is open; *ret* is the set of retired researchers; *hiredRet* is the set of retired researchers that has been hired by CONICET's to continue with their duties; and *inIntranet* is the set of researchers that have an user account in CONICET's Intranet.

Two interesting conditions of this election system is that voters can vote only candidates of their same KA ($\{myKA\, e?\} = myKA(\!| C? |\!)$); but voters must vote candidates all from different GR ($\#C? = \#(myGR(\!| C? |\!))$). In Z $(\!|\_|\!)$ is the relational image of a set under a function; and # is the cardinality operator.

Under all these conditions the system shall add the voter to the set of people who has already voted and the vote to the logical ballot box. This is expressed by the post-conditions:

$$\begin{array}{l} voted' = voted \cup \{e?\} \\ votes' = votes \frown \langle C? \rangle \end{array}$$

where $\cup$ is the classical union set operator and $\frown$ concatenates two sequences.

Recall that *Vote* is defined as the disjunction of eight schemas where *VoteOk* describes the conditions when a vote is successfully issued and all the others represent error conditions. Below we include one of these "error" schemas:

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

Note that this schema include the expression ΞES meaning that it will not change the value of any of the variables declared in *ES*. So *AlreadyVoted* is saying that if the person willing to issue a vote ($e?$) has already voted ($e? \in voted$) then nothing will change.

All the operations defined in the model share the same structure (i.e. a schema defined as the disjunction of some schemas one of which represents the successful case and the rest represent erroneous situations).

2.2 State Invariants

A state invariant is a predicate, depending on state variables, that is true of every state of the state space. 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 follows[4]:

Invariants

ES

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

and then we include a proof obligation such as the following for each operation in the model:

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

where *Vote* is the schema defined in the previous section and $Invariants'$ is the same than *Invariants* but where all the state variables are decorated with a prime. Then if theorem VotePreservesInvariants is proved it means that if the state invariants hold before executing *Vote*, then they will hold right after its execution. In other words, *vote* preserves the invariants.

If invariants are codified in this way then all preconditions must be explicit [8, 2, 9] and Fastest becomes more efficient. In a sense, this way of writing invariants changes "calculate the full precondition of an operation" by "discharge a proof obligation"—see [4, Sect. 2.7] for a few more details.

Some of the predicates in *Invariants* are the formalization of CONICET rules. For example, see the seventh predicate in *Invariants*. CONICET established that after the end of the period reserved for endorsements only candidates who have got at least 20 endorsements become firm candidates. In the model *fEndors* represents the final date of the endorsements period. In Z: dom is the domain of a function; ran is the range of a function; and if f is a function and A is a set, then $f \rhd A$ (called range restriction) is the function obtained from f by removing all the ordered pair whose second component belongs to A. Therefore:

$$fEndors < today \Rightarrow firmCand = \mathrm{dom}(endors \rhd \{A : \mathrm{ran}\ endors \mid 20 \leq \#A\})$$

says that if the current date (*today*) is after the end of the endorsement period (*fEndors*), then the set of firm candidates (*firmCand*) must be the domain of

[4] Just take a look at it for now; later we give some details about it.

endors restricted to those who got at least 20 endorsements. Recall that *endors* is a state variable declared in schema *Endorsements* as $RSCH \pfun \mathbb{F}\, RSCH$, and whose meaning is that $endors(e)$ is the set of researchers who endorsed e for the current election.

Other invariants arise from the model itself. For instance, the domains of *cand* and *endors* must always be the same because even a candidate who got no endorsements is represented as having an empty set of endorsements.

3 Proving Properties of the Specification

The Z specification was verified under the Z/EVES system [10]. Therefore, we divided the task in two steps: a) discharging all the proof obligations automatically generated by Z/EVES; and b) proving that all the operations preserve all the invariants. Z/EVES generates a proof obligation, called domain check, every time a partial function is applied to an argument[5]. 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:

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

where $C? \in \mathrm{dom}\,\#$ is trivial to prove since the domain of $\#$ is $\mathbb{F}\, X$ for any type X; but $miRE(\!|\, C? \,|\!) \in \mathrm{dom}\,\#$ is more difficult 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) [11] proposed by Freitas [12].

The second step in verifying the specification consisted in proving that all operations preserve the state invariants described in schema *Invariants* shown in page 8. We did this by proving a theorem like VotePreservesInvariants shown at page 8. 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 schemata and there are many invariants. But, on the other hand, many of this 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:

[5] Actually, Z/EVES also generates proof obligations for definite descriptions (μ-terms), but we did not use them.

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

as an intermediate theorem for the operation representing a voter endorsing a preliminar candidate. Note that in this case we prove that only the schema corresponding to the successful case (*EndorseOk*) preserves only the fifth invariant (see the fifth predicate in schema *Invariants* at page 8). There are three more theorems like this one for the *Endorse* operation. Once all these theorems are proved, the main theorem (i.e. that establishing that *Endorse* preserves all the invariants) can be proved easily by iterating over all the schemata that define the operation and all the invariants. A similar approach was applied for all the operations.

Although the ZMT includes a number of theorems about all the mathematical theories supported by the Z notation and the Z/EVES system uses them for proofs, from time to time it is necessary to prove some new result about mathematics. In this project we needed to prove five of these theorems. For example, we proved that the relational image of a finite set through a partial function is a finite set:

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

which was necessary to prove two domain checks. As an another example we can mention the following theorem which is a particular form of the applyInRanPfun theorem of the ZMT and was used in the proofs confirming that operation *PublishCandidates* preserves the fourth and fifth invariants[6]:

theorem applyInRanSinglePfun $[X, Y]$
$\forall A : \mathbb{P}\, X;\ B : \mathbb{P}\, Y \bullet \forall f : A \pfun B \bullet \forall x : \text{dom}\, f;\ y : Y \bullet \text{ran}\, f = \{y\} \Rightarrow f\, x = y$

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 in two verifications activities: a) inspecting the code to see if it refines

[6] *PublishCandidates* is the first step during the election process in which CONICET publish the list of candidates looking for endorsements.

the specification; and b) run some tests cases derived from the specification. Clearly, the first activity is static while the second is dynamic thus we made use of two fundamental verification strategies. In this section we explain how we used the specification to inspect the source code of the application.

Before of showing how we inspected the code we will give some insight on the implementation of the voting system. As we have said in the introduction, the application was developed by CONICET's System Department. They decided to program it as a Grails application. Grails [13] is web application framework for the Java Virtual Machine which in turn takes advantage of the Groovy programming language [14]. 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 data structures and variables that refine the state variables declared in schemata *Endorsements* and *Votes* (see page 4) and the input variables declared in he operation schemata like *Vote*. 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, as is said in the requirements document, the system stores the name of each candidate plus his or her CV and photo, but we decided to abstract away these details in the specification. As another example, the requirements document, and thus the specification, does not mention data encryption at all, but it is used in the implementation.

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.

Table 1. Implementation variables refining specification variables.

We will comment on the first three rows of Table 1. 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. In turn, variable *voted* is implemented by saving the system date when the voter votes in his or her database record. Voters are hold in a table named `Elector` one of whose fields is the identification of the voter in the system, and the other is the date when the voter has voted; this table acts as the electoral

roll. *votes* is refined by a table in the same database named `EncryptedVote`. The table's name reflects the fact that, due to security considerations, developers decided to encrypt each vote and then persist it in the table. Each of the elements stored in `EncryptedVote` is the result of encrypting an instance of class `Vote`. In turn, `Vote` stores not only the name of candidates but also their GR, although that could be traced through the data model.

Once we had clear how state variables were implemented we identified all the major operations of the specification. This involves to find the relation between the signature of operations at the specification level and subroutines in the implementation, and check whether they match or not. We will analyze the case of a person issuing his or her vote—i.e. *Vote*. The code dealing with voting is divided in several files two of which, `VoteController.groovy` and `VoteService.groovy` are the main ones. Control flow starts in `VoteController.groovy`. There the system presents to the user all the candidates, he/she can select some of them and press a button. Then, a method, called `issue()`, in `VoteService.groovy` is called and if everything is right the vote is saved in the database. If one looks at the first method executed in `VoteController.groovy` then its interface mismatch that of *Vote* in the specification, because it does not wait any input parameters. The refinement of the interface of *Vote* is found in the signature of `issue()` which is called from `VoteController.groovy`.

The last step during the code inspection was to check whether all the pre and postconditions 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. We will continue with the analysis of the operation *Vote*. Firstly, we labeled each pre and post-condition in the specification as can be seen in schema *VoteOk* at page 2.1. These identifiers can latter be used to annotate the implementation as a means to link both representations. Secondly, we used the information gathered in the two previous steps to focus the inspection on specific implementations units. For example, in order to inspect the implementation of *Vote* we read code in `VoteController.groovy` and `VoteService.groovy`, and we looked for the refinement of variables *voted*, *firmCand*, *votes*, etc. Every time we found the implementation of a pre or post-condition we annotated the program as follows:

```
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
.....................................................
```

Note 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.

In `VoteService.groovy` the method `record()` receives the ID of the voter and the encrypted vote and, among other things, checks again if the user can vote and, if not, it writes the system date in the record of the electoral roll corresponding to the user:

```
private record(String userId, byte[] encryptedVote) {
.....................
//Begin Vote::Pre-3, Vote::Pos-3
Elector elector = Elector.findByUserId(userId)

if (elector.voteDate)
  throw new RuntimeException("User already voted.")
//End Vote::Pre-3, Vote::Pos-3
.......................
//Begin Vote::Pos-1
elector.voteDate = new Date()
//End Vote::Pos-1
}
```

The reader can see that we repeated the annotation in this piece of code, and we added an annotation regarding the first postcondition of schema *VoteOk*.

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. For example, note that we annotated the beginning of $\mathsf{Pre}-3$ in `VoteController` after the value of `elector` is set (see the `def` instruction right above the annotation). Clearly, if this variable is incorrectly assigned the

precondition will be incorrectly implemented. Therefore, if during maintenance the assignment sentence is changed or a new assignment to `elector` is added, a convenient IDE can detect that this variable is used in a critical precondition and warn the programmer.

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 true in general but 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 [15, 4, 5, 16–18], here we will show, by means of a running example, how we applied them to generate test cases. The TTF and Fastest are used for unit and functional testing.

Assume the specification has been loaded in Fastest. Consider schema *Vote* (shown at page 5) and its input space. The input space of an operation is a Z schema declaring all the input and state variables of the operation and not restricted by a predicate. First, we partition the input space of *Vote* by applying so-called testing tactics. For example, we can apply Disjunctive Normal Form (DNF) in which case Fastest generates nine test conditions, including the following three (where $Vote^{IS}$ is the schema representing the input space of *Vote*):

$Vote_1^{DNF}$

$Vote^{IS}$

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

$Vote_3^{DNF}$

$Vote^{IS}$

$fVote < today$

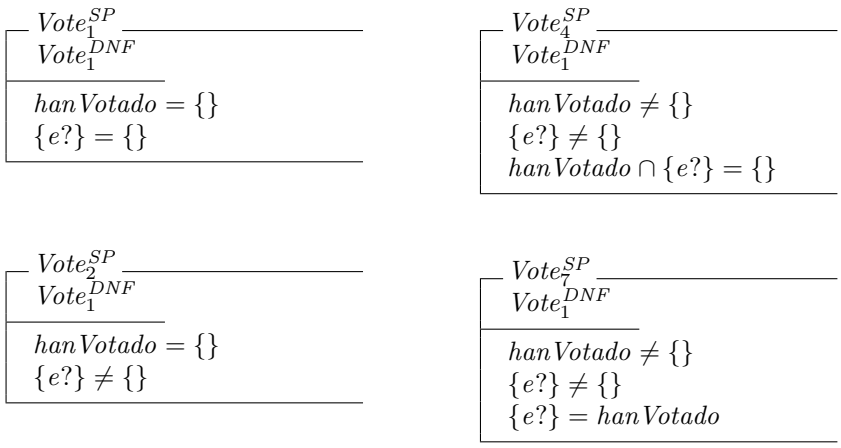
$Vote_5^{DNF}$

$Vote^{IS}$

$e? \in voted$

Note that these schemata 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 by applying other testing tactics. 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:



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 [6]. 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.

More testing tactics can be applied to further partition the input space of the operation. For instance, another standard partition can be applied to $\frown$, in $votes' = votes \frown \langle C? \rangle$, to partition all the satisfiable children of $Vote_1^{DNF}$. Test conditions can be arranged in a so-called testing tree by following the schema inclusions. Partitioning is automatically done by Fastest while the elimination of unsatisfiable test conditions is semi-automatic.

Once the engineer is done with partitioning he or she can generate test cases from the leaves of the testing tree. In the TTF generating a test case means finding constants that satisfy the predicate of the test condition. In Fastest this is done by calling a satisfiability algorithm[7]. As with the elimination of unsatisfiable test conditions, this algorithm is necessarily incomplete and thus semi-automatic. However, according to our experiments, in average the tool automatically finds test cases for at least 80% of the satisfiable test conditions; the rest must be calculated by the user. Below we show two typical test cases generated by Fastest ($rsch0$ is a constant of type $RSCH$ identifying a particular researcher).

[7] The algorithm currently implemented by Fastest has been described in [4] but is being replaced by one based on the $\{log\}$ (setlog) tool, see [19] for details.

$Vote_2^{TC}$

$Vote_2^{SP}$

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

$Vote_4^{TC}$

$Vote_4^{SP}$

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

These test cases, however, are Z terms that cannot be executed by the application. Usually these test cases are called abstract test cases given that they are at the same level of abstraction of the specification. Hence, we gave developers precise instructions on how to write JUnit [20] test cases from the Z test cases, and we translated some of them as examples. For the translation 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.

It is important to remark that, although we had to write the JUnit test cases by hand, Fastest and the TTF were very helpful because without them we also would have to design the test cases. Test case design is the most time-consuming and error-prone activity of testing [21]. JUnit users need to design the test cases by hand and then write them in JUnit. Furthermore, in [5] we proposed FTCRL, a method to refine abstract test cases generated by Fastest. In other words, FTCRL would semi-automate the translation from Fastest test cases into test cases similar to JUnit test cases. However, the implementation of the method is not ready yet.

6 Discussion

The Z specification presented here is about 450 lines of Z code (LaTeX mark-up) 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—i.e. each of them proves that the main schema of each operation preserves one invariant. 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 we new 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. Test case generation required to translate the specification from Spivey's de-facto standard into the ISO standard, although, in this case, it was very simple. 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 the time consumed during 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 (i.e., their endorsements, KA and GR) 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 Sect. 7. However, our project presents crucial differences with many, if not all, of them. These other projects are performed entirely, including the programming stage, by experts on formal verification—many of them are either PhD students within first-class research groups or people already holding a PhD. Furthermore, in all these projects the implementation is developed once the specification has been written and proved correct. Indeed, many formal techniques have been thought to be applied under the assumption of a correct specification. 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. It is true, however, that the verification was performed by researchers. In summary, we think that the project presented here is closer to the way the software industry works, making it appealing to evaluate the applicability of formal methods. It remains as a challenge to run a lager project but in a similar context.

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 [22]. This project was an experiment performed by the NSA to prove that formal methods are cost-effective in real-world software. Altran Praxis from UK was finally hired for the job. They wrote a Z specification that was latter 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 [23]. Altran Praxis has used these technologies in many projects of different application domains.

Cristiá, Frydman 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 [18]. In all these projects the TTF and Fastest were applied to generate test cases.

Frydman and her colleagues have combined Z with DEVS [24] for the validation of discrete event systems via simulation and formal methods [25, 26]. Object-Z has been used in the formalization of the Web Service Modeling Ontology for the Semantic Web [27] and also Z and Z/EVES were applied in the same domain [28]. 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 [29] the authors combined Z and CSP [30] to provide a formal specification for the Audited Credential Delegation architecture which would help virtual organization 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 [31] 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 [32], 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 where Z was the specification language. Khan et al. [28] and Zafar and his colleagues [32] 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 mean to gain confidence on its correctness. For example, in [33] the authors used this tool to prove 40 theorems about feature modeling. Amalio an 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 [34] and for the UML modeling notation [35]. Dong and Wang [36] explore the benefits of using Z/EVES to detect inconsistencies in semantic web services, although it seems that they did not prove a reasonable amount of properties. Yuan and others [37] 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, 39] and a POSIX file store [40], 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 [6].

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 [41] annotate an application with ESC/Java [42] 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 latter analyzed by the SPARK Examiner [43]. 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 [44–47]

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 [48]. Another ad-

vanced tool that was applied to large, safety-critical, embedded, real-time software is presented by Blanchet and his colleagues [49]. 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 [50, 51]. 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 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 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) can be the basis of a verification methodology for mission critical applications whose verification is requested once the implementation is finished.

References

1. Cristiá, M., Frydman, C.S.: A functional verification of a web voting system. In Murgante, B., Misra, S., Rocha, A.M.A.C., Torre, C.M., Rocha, J.G., Falcão, M.I., Taniar, D., Apduhan, B.O., Gervasi, O., eds.: ICCSA (1). Volume 8579 of Lecture Notes in Computer Science., Springer (2014) 640–655
2. Spivey, J.M.: The Z notation: a reference manual. Prentice Hall International (UK) Ltd., Hertfordshire, UK, UK (1992)

3. ISO: Information Technology – Z Formal Specification Notation – Syntax, Type System and Semantics. Technical Report ISO/IEC 13568, International Organization for Standardization (2002)
4. Cristiá, M., Albertengo, P., Frydman, C.S., Plüss, B., Rodríguez Monetti, P.: Tool support for the Test Template Framework. Softw. Test., Verif. Reliab. **24**(1) (2014) 3–37
5. Cristiá, M., Hollmann, D., Albertengo, P., Frydman, C.S., Monetti, P.R.: A language for test case refinement in the Test Template Framework. In Qin, S., Qiu, Z., eds.: ICFEM. Volume 6991 of Lecture Notes in Computer Science., Springer (2011) 601–616
6. Cristiá, M., Albertengo, P., Rodríguez Monetti, P.: Pruning testing trees in the Test Template Framework by detecting mathematical contradictions. In Fiadeiro, J.L., Gnesi, S., eds.: SEFM, IEEE Computer Society (2010) 268–277
7. Jackson, M.: Software requirements & specifications: a lexicon of practice, principles and prejudices. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA (1995)
8. Jacky, J.: The way of Z: practical programming with formal methods. Cambridge University Press, New York, NY, USA (1996)
9. Saaltink, M.: The Z/EVES 2.0 User's Guide. Ora Canada. (1999)
10. Saaltink, M.: The Z/EVES system. In Bowen, J.P., Hinchey, M.G., Till, D., eds.: ZUM. Volume 1212 of Lecture Notes in Computer Science., Springer (1997) 72–85
11. Saaltink, M.: The Z/EVES mathematical toolkit version 2.2 for Z/EVES version 1.5. Technical report, ORA Canada (1997)
12. Freitas, L.: Z/Eves extended Z toolkit. Technical report, University of York (2004)
13. SpringSource: Grails – The search is over last access: February 2013.
14. SpringSource: Groovy – A dynamic language for the Java platform last access: February 2013.
15. Stocks, P., Carrington, D.: A Framework for Specification-Based Testing. IEEE Transactions on Software Engineering **22**(11) (November 1996) 777–793
16. Cristiá, M., Plüss, B.: Generating natural language descriptions of Z test cases. In Kelleher, J.D., Namee, B.M., van der Sluis, I., Belz, A., Gatt, A., Koller, A., eds.: INLG, The Association for Computer Linguistics (2010) 173–177
17. Cristiá, M., Frydman, C.S.: Extending the Test Template Framework to deal with axiomatic descriptions, quantifiers and set comprehensions. In Derrick, J., Fitzgerald, J.A., Gnesi, S., Khurshid, S., Leuschel, M., Reeves, S., Riccobene, E., eds.: ABZ. Volume 7316 of Lecture Notes in Computer Science., Springer (2012) 280–293
18. 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, IEEE Computer Society (2011) 128–137
19. Cristiá, M., Rossi, G., Frydman, C.: $\{log\}$ as a test case generator for the test template framework. Technical report, Dipartimento di Matematica, Università di Parma (2012)
20. Saff, D.: JUnit.org – Resources for Test Driven Development. Available at: `http://junit.org/`. Last access:
21. Jones, C., Bonsignour, O.: The Economics of Software Quality. Addison-Wesley (2011)
22. 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)

23. 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)
24. Zeigler, B.P., Kim, T.G., Praehofer, H.: Theory of Modeling and Simulation. Academic Press, Inc., Orlando, FL, USA (2000)
25. 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 (descember 2009)
26. 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 (EMSS09), Tenerife - Canary Islands, Spain (september 2009)
27. Wang, H.H., Gibbins, N., Payne, T.R., Redavid, D.: A formal model of the semantic web service ontology (WSMO). Information Systems **37**(1) (2012) 33 – 60
28. Khan, S.A., Hashmi, A.A., Alhumaidan, F., Zafar, N.A.: Semantic web specification using Z-notation. Life Science Journal **9**(4) (2012)
29. 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. Volume 16 of EPTCS. (2009) 41–50
30. Roscoe, A.W.: The Theory and Practice of Concurrency. Prentice Hall PTR, Upper Saddle River, NJ, USA (1997)
31. Gomes, A.O., Oliveira, M.V.: Formal specification of a cardiac pacing system. In: Proceedings of the 2nd World Congress on Formal Methods. FM '09, Berlin, Heidelberg, Springer-Verlag (2009) 692–707
32. 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)
33. 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, Washington, DC, USA, IEEE Computer Society (June 2005) 303–312
34. Amálio, N., Stepney, S., Polack, F.: A formal template language enabling metaproof. In Misra, J., Nipkow, T., Sekerinski, E., eds.: FM. Volume 4085 of Lecture Notes in Computer Science., Springer (2006) 252–267
35. Amálio, N., Stepney, S., Polack, F.: Formal proof from uml models. In Davies, J., Schulte, W., Barnett, M., eds.: ICFEM. Volume 3308 of Lecture Notes in Computer Science., Springer (2004) 418–433
36. Dong, J.S., Sun, J., Wang, H.H.: Z approach to semantic web. In George, C., Miao, H., eds.: ICFEM. Volume 2495 of Lecture Notes in Computer Science., Springer (2002) 156–167
37. 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. Volume 4318 of Lecture Notes in Computer Science. Springer Berlin Heidelberg (2006) 196–210
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. Volume 4281 of Lecture Notes in Computer Science. Springer Berlin Heidelberg (2006) 15–34
39. Freitas, L., Woodcock, J.: Mechanising mondex with z/eves. Formal Aspects of Computing **20** (2008) 117–139

40. 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** (2007) 3–14
41. 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. FME '02, London, UK, UK, Springer-Verlag (2002) 272–289
42. Compaq Systems Research Center: Extended static checking for Java last access: February 2013.
43. King, S., Hammond, J., Chapman, R., Pryor, A.: Is proof more cost-effective than testing? IEEE Trans. Software Eng. **26**(8) (2000) 675–686
44. Bartetzko, D., Fischer, C., Möller, M., Wehrheim, H.: Jass - java with assertions. Electr. Notes Theor. Comput. Sci. **55**(2) (2001) 103–117
45. Jacobs, B.: Weakest pre-condition reasoning for java programs with jml annotations. J. Log. Algebr. Program. **58**(1-2) (2004) 61–88
46. 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) (2004) 89–106
47. Frade, M.J., Pinto, J.S.: Verification conditions for source-level imperative programs. Computer Science Review **5**(3) (2011) 252 – 277
48. Berdine, J., Cook, B., Ishtiaq, S.: Slayer: Memory safety for systems-level code. In Gopalakrishnan, G., Qadeer, S., eds.: CAV. Volume 6806 of Lecture Notes in Computer Science., Springer (2011) 178–183
49. Blanchet, B., Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A., Monniaux, D., Rival, X.: The essence of computation. Springer-Verlag New York, Inc., New York, NY, USA (2002) 85–108
50. 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) –
51. 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, Springer (2010) 145–153