

Formal object-oriented development of a voting system test oracle

J. Paul Gibson · Jean-Luc Raffy · Eric Lallet

Received: 30 April 2011 / Accepted: 1 September 2011 / Published online: 29 September 2011
© Springer-Verlag London Limited 2011

Abstract We report on the formal development of a test oracle for an electronic-voting system, detailing our approach which was based on a *lightweight* integration of Event-B, UML, and Java. This lightweight integration focuses on a coherent development process rather than on formal integration at the model/language level. We then briefly review alternative development approaches involving the use of JML and OCL. We conclude that the use of the OCL would offer few benefits, if any, in our formal development process.

Keywords Event-B · UML · Java · JUnit

1 Introduction

In this paper, we consider the problem of developing a test oracle for an electronic-voting system. This problem arose out of a national research project *Sécurité et Audit du Vote Électronique*¹ whose objective was the development of a novel distributed e-voting system that would facilitate voters to vote at any official voting station. The system also enforces anonymity and limited voter verifiability—where a voter can check that their vote has been counted [6]—through a voting protocol based on homomorphic encryption [9].

We have already reported on the design of the distributed architecture of the SAVE system with respect to meeting quality of service issues [12], and the complex issue of feature interactions in e-voting systems [13]. Further, we have also

reported on the need for a mix of formal methods in the modelling of e-voting systems [14].

As we approached the end of the project, we realised that our prototype system had yet to be verified with respect to the accuracy of the counting process. The transferable vote counting algorithm had already been formally specified, but this specification had been used only in helping the programmers to understand the counting requirements rather than in any formal design or verification.

The prototype had been developed in Java and had a limited testing interface/harness. It was beyond the scope of the project to re-engineer the count code, which involved the use of very complex encryption mechanisms. Thus, our only reasonable choice was to develop a test oracle whose sole purpose was to verify the result of the count generated by the prototype system. Given that the accuracy of the count is considered critical, we decided that we should develop the oracle as formally as possible.

As we had already modelled the vote count using Event-B [1] and decided that our test oracle should be written in Java (to simplify the connection with the existing prototype and test harness), the open question was how best to move from requirements to implementation of the oracle as formally as possible. We decided to adopt the UML to model structural properties of the problem, and design decisions which were made during the development. Rather than adopting new tools or formalisms that could better integrate our three languages—Event-B, UML and Java—our initial goal was to find a useful methodological integration. Then, any weaknesses in our methodology would help us to better understand our software engineering development requirements in order to make a more informed judgement as to whether we would

J. P. Gibson (✉) · J.-L. Raffy · E. Lallet
T & MSP, Evry, France
e-mail: paul.gibson@it-sudparis.eu

J.-L. Raffy
e-mail: jean-luc.raffy@it-sudparis.eu

E. Lallet
e-mail: eric.lallet@it-sudparis.eu

¹ SAVE project URL: <http://www.projet-save.fr>.

be better adopting a modelling approach based on a strong(er) integration of our chosen formalisms.

2 SAVE test oracle requirements

In this section, we consider the move from requirements to high level design.

2.1 Requirements

The following natural language description of the test oracle requirements was used to identify the main classes and the relationships between them: The test system is made up of three main components: the interface, the ballot module and the tabulator. After each voter has validated their vote as recorded on a ballot at the interface, the completed ballot is transferred to the module. Once the election is completed, the transfer of votes from the interface to the ballot module is forbidden. The tabulation process starts by annotating the copied ballots for counting purposes.

Further, safety requirements were also recorded: only valid ballots are stored in the system during the voting process. Ballots cannot be added or removed in transfer during the tabulation. Moving ballots between system components do not change the record of the voter’s intent. The tabulation process cannot change or invalidate ballots.

2.2 High-level design

Our first development step is to use UML to model the high-level design structure of our test oracle tabulator (see Fig. 1).

The next step in the development process is to identify the requirements for each of the classes in our design and to for-

malise them. The following natural language text describes the requirements for one of the key classes in the test system, namely the ballots: a ballot consists of a non-empty list of candidates. Our interface limits the number of candidates to 20. A completed ballot associates preferences with candidates. The number of preferences allocated must be non-zero and not bigger than the number of candidates. Only preferences in the range 1 up to the number of preferences are allocated. Preferences cannot be assigned to candidates outside the range 1 up to the number of candidates. No two candidates can be allocated the same preference.

Functional requirements, specific to ballots, were stated as: for each ballot, we must be able to see: the number of candidates, the number of preferences, the candidate allocated to every preference and the preference allocated to each candidate. When a candidate has no preference, then their associated preference value is 0. When a preference has no candidate, then the associated candidate value is 0, otherwise it is the unique integer identifier used in the enumeration of candidates.

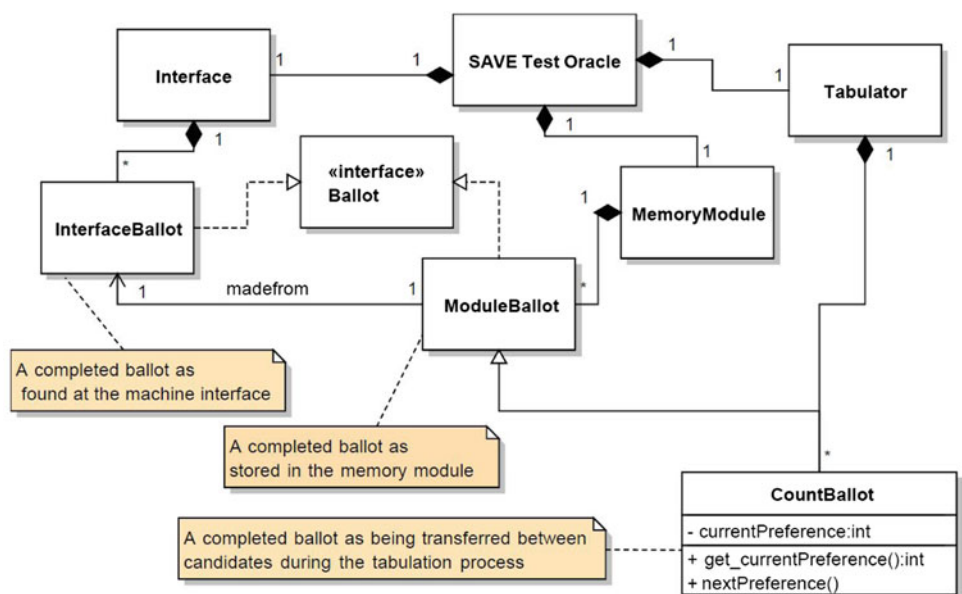
3 The lightweight integration approach

In this section, we compare different approaches to integrating different modelling languages, and outline why we consider our approach to be lightweight.

3.1 Integrating (formal) methods: the compromises

We propose that there are three main approaches to integrating modelling languages during software development (see Fig. 2). We argue that such integration is necessary because no single language is adequate for modelling during all stages of software development.

Fig. 1 The high-level design for the tabulator test oracle



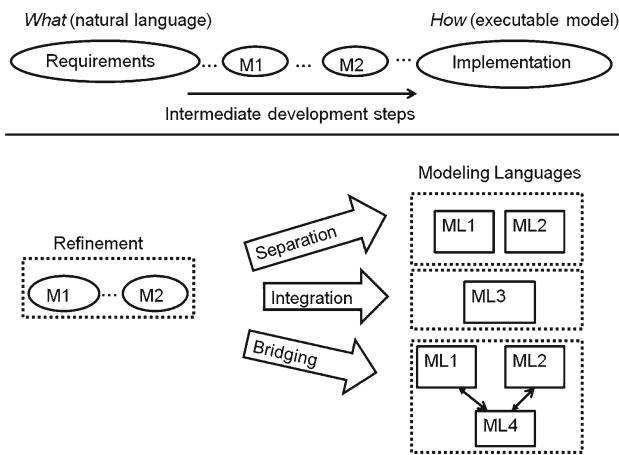


Fig. 2 Integrating (formal) methods: the compromises

Development is considered to be a sequence of refinements, which transform an abstract model of the system into a more concrete model (maintaining correctness). The problem is that changing levels of abstraction often, but not always, requires changing modelling language. We categorize three different types of refinement step. First, **separation** is where each level of abstraction is modelled using a separate language, but these are close enough such that there are clear, well-defined relationships between models written in these different languages. Second, **integration** is where the same modelling language can be used at different levels of abstraction. Finally, **bridging** is where each level of abstraction is modelled using a separate language, but these are not close enough such that there are clear, well-defined relationships between models written in these different languages. Thus, a bridging model language must be used to cross the semantic gap.

In the case study reported in this paper, we propose to “refine” Event-B models into Java code. Since the largest semantic gap between our two languages is the use of object-oriented concepts in one, but not in the other, we chose to adopt the UML as our bridging language. In Sect. 7, we consider alternatives to using UML to bridge this gap.

The integrated approach has the enormous advantage of simplifying verification (coherency) between models at different levels of abstraction. However, such approaches are not feasible for the whole development cycle; when they do span large parts of development, they often lead to either very rich languages that are difficult to master or languages that are not well adapted for the different abstraction levels.

The separated approach is an ideal situation: each language is perfectly suited for its level of abstraction and there are formal relations between models in the different languages that support automated verification techniques.

The bridging approach uses an intermediate model to bridge the semantic gap. Note that such an intermedi-

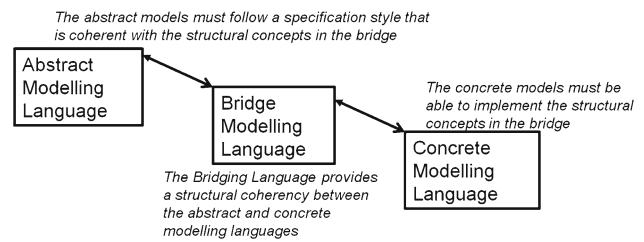


Fig. 3 Lightweight: methodological integration

ate model may (and probably does) exist in the separated approach, but the difference is concerned with its visibility to the developers. In the bridging approach, the developers must work directly with the bridging models. They cannot be completely generated automatically; neither by refining the more abstract models nor through abstract interpretation of the more concrete models.

3.2 A lightweight methodological integration

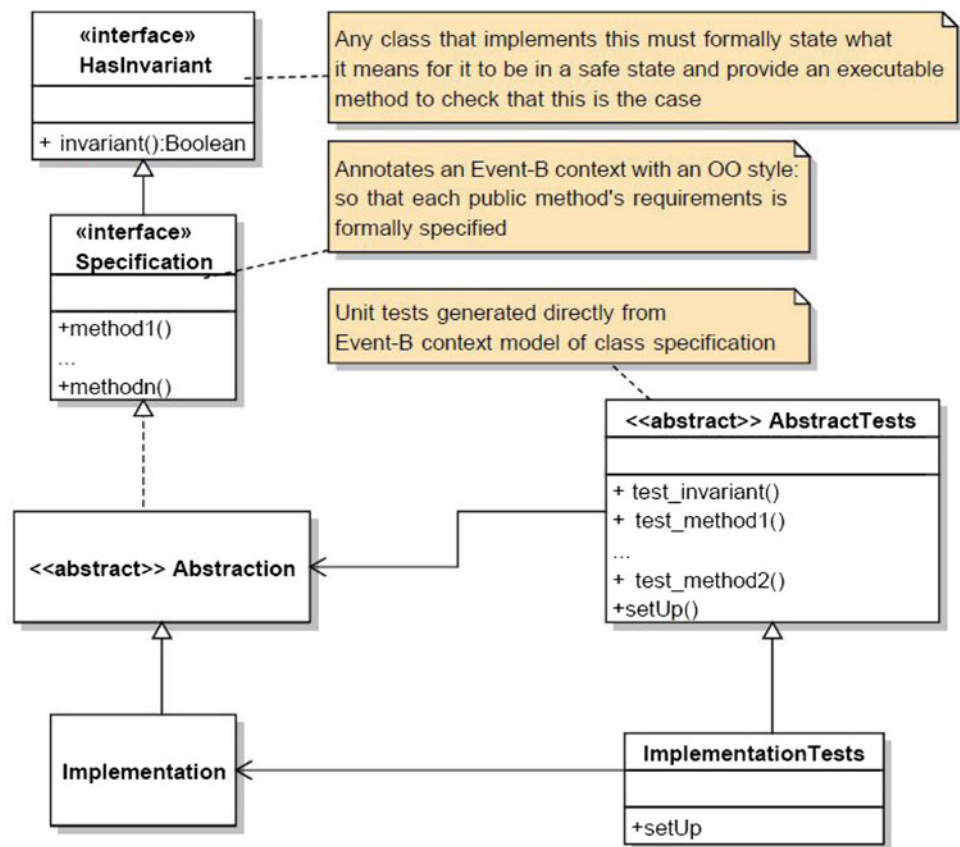
We refer to our integration as being *lightweight* because it is based on following a development process that depends on a particular style of modelling rather than on a fully formal integration. Figure 3 shows how one can bridge between modelling languages at different levels of abstraction by using a modelling style rather than a more formal language integration. In our development process, we focus on using object-oriented concepts in our bridge.

3.3 Development process *Design Pattern*

In Fig. 4, we illustrate how the UML, Java and Event-B are loosely integrated in our development process.

The natural language requirements are used to identify the classes, their responsibilities, and the relations between them. These are then documented in a UML class diagram. Every class is then developed using the generic development design pattern. Class responsibilities are specified using an interface specification, i.e a set of methods whose functional requirements are formalised in an Event-B class model—a context written using an OO style (see Sect. 3.4). Every class must also specify an invariant method which states—in terms of the public responsibilities of the class—when class instances are in a safe state. For classes with a subset of external responsibilities which can be implemented abstractly through the use of methods defined in the interface specification, these implementations (methods) are grouped together in a single abstract class for re-use. Every class has a set of unit tests whose code is derived directly from the Event-B specification. These tests are implemented abstractly in an `AbstractTests` class. Any class that claims to correctly implement the abstraction must pass all abstract unit tests.

Fig. 4 The generic development design pattern for all classes in the UML high-level design



3.4 Event-B contexts in an OO style

Event-B contexts can be used to model abstract data types in an algebraic style. Following the formal object-oriented approach to requirements modelling as advocated by Gibson [11], we address three key issues when modelling a class of behaviours:

- **A means of categorising entities into classes of behaviour:** we use Event-B contexts with sets representing classes.
- **A means of recording the external interface of a class, so that all methods can be statically 'type checked' for correctness:** we define constant operation signatures in the Event-B.
- **A means of defining the behaviour associated with each operation:** we use Event-B axioms.

Three different types of methods/signatures—as proposed in [11]—are supported in our Event-B modelling style when specifying the behaviour of a UML `class`:

- **ACCESSORS** methods with return values, but which do not change the state of the object being called have signatures

of the form: $method(p_1, \dots, p_n) : result$. These are modelled in Event-B using an axiom of the form: $method \in class \times p_1 \times \dots \times p_n \mapsto result$. In a queue of integer values, for example, we would specify an accessor to check if the queue was empty as follows: $is_empty \in queue \mapsto BOOL$

- **TRANSFORMERS** methods with no return values, but which do change the state of the object being called have signatures of the form: $method(p_1, \dots, p_n)$. These are modelled in Event-B using an axiom of the form: $method \in class \times p_1 \times \dots \times p_n \mapsto class$. Continuing with the queue example, we would specify a transformer to push an element on to the queue as follows: $push \in queue \times \mathbb{N} \mapsto queue$
- **DUALS** methods with return values that also change the state of the object being called have signatures of the form: $method(p_1, \dots, p_n) : result$. These are modelled in Event-B using an axiom of the form: $method \in class \times p_1 \times \dots \times p_n \mapsto class \times result$. For example, we would specify a dual to pop an element off the queue as follows: $pop \in queue \mapsto queue \times \mathbb{N}$

When the behavioural requirements are specified for only a subset of parameter values, then the method relations are

specified using partial functions. (In our Java implementation, we chose to implement such cases using exceptions.)

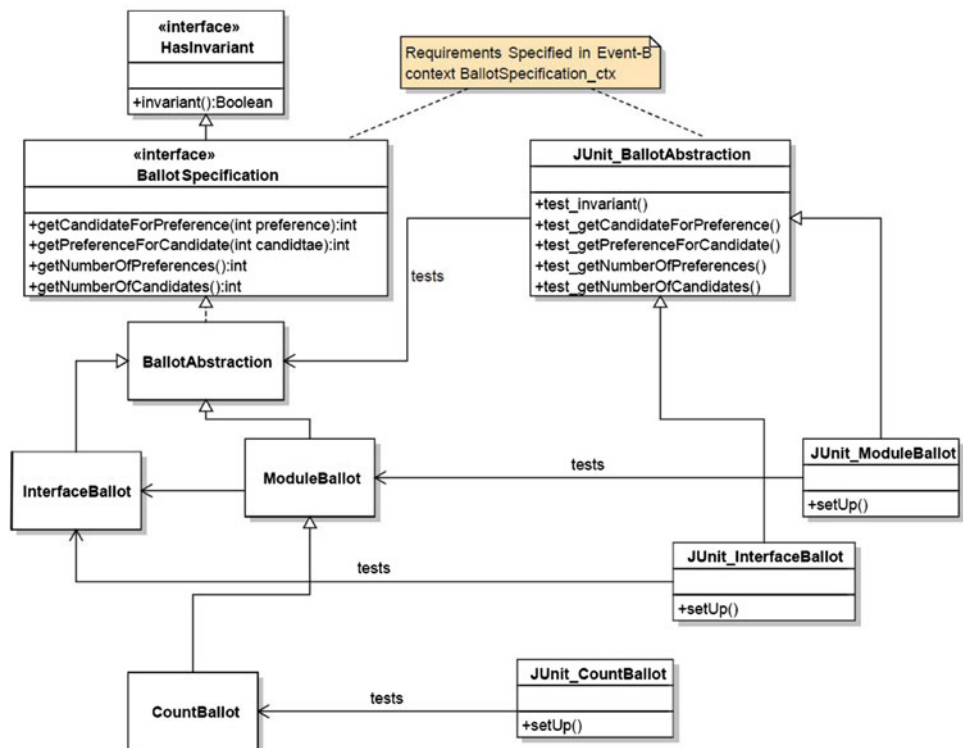
Our lightweight approach uses Event-B contexts to specify abstract class requirements and facilitates modelling more concrete design and implementation details using Event-B machines, and their refinements. The move from requirements to design can be automated through the generation of an abstract Event-B machine corresponding to a class as specified in a context. This is similar to the approach in [11].

In the voting case study, the best illustration of this is in the different implementations of the ballot requirements. In the interface, we chose to implement the ballot as an array of candidates mapped to preference values; whilst in the memory module, we chose to implement the ballot as an array of preferences mapped to candidate values. The use of Event-B facilitates formal verification of the equivalence of these two different implementations (with respect to the abstract requirements).

4 Applying the approach to the test oracle

In the following subsections, we consider the formalisation and verification of the natural language safety requirements as noted in Sect. 2. Due to lack of space, we provide details of only the first of these requirements concerned with the validity of votes and sketch a brief overview of how the other three requirements are modelled.

Fig. 5 Instantiation of generic development design pattern for ballots



4.1 Only valid ballots are stored in the the system during the voting process

Once the Event-B specifications have been completed, we instantiate the abstract design pattern with the concrete classes in our initial design. In this paper, we focus on the two classes which represent a ballot at the interface and in the memory module (see Fig. 5).

The Event-B axioms for the signature of the abstract ballot class are given in Fig. 6.

The `nextPreference()` method (see Fig. 1) transfers a ballot to another candidate during tabulation. The proof that this method respects the abstract invariant is trivial (since the only state attribute that is changed is `currentPreference`) and is discharged automatically by the RODIN tool when modelled using Event-B.

The next step is to define the invariant requirements using Event-B axioms. This is illustrated in Fig. 7, where the invariant has been decomposed into the conjunction of seven requirements, of which we show four.

4.2 Ballots cannot be added or removed in transfer during the tabulation

The specification and verification of this requirement is trivial in Event-B. Currently, we model this requirement as an invariant on the number of ballots. The only state that changes during tabulation events is the candidate for which a ballot is

```

axm1 : getNumberOfCandidates ∈
      Ballots → MIN_NUMBER_OF_CANDIDATES ..
              MAX_NUMBER_OF_CANDIDATES
      Ballot Method Signature: Accessor
axm2 : getNumberOfPreferences ∈
      Ballots → 1 .. MAX_NUMBER_OF_CANDIDATES
      Ballot Method Signature: Accessor
axm3 : getPreferenceForCandidate ∈
      (Ballots × (1 .. MAX_NUMBER_OF_CANDIDATES))
      →
      0 .. MIN_NUMBER_OF_CANDIDATES
      Ballot Method Signature: Accessor
axm4 : getCandidateForPreference ∈
      (Ballots × (1 .. MAX_NUMBER_OF_CANDIDATES))
      →
      0 .. MAX_NUMBER_OF_CANDIDATES
      Ballot Method Signature: Accessor
axm5 : invariant ∈ Ballots → BOOL
      Ballot Method Signature: Accessor

```

Fig. 6 Abstract ballot signature axioms

currently assigned. Thus, the number of ballots is sure never to change, as we refine the tabulation machine to provide further implementation details.

4.3 Moving ballots between system components do not change the record of the voter's intent

A recurring reported problem with e-voting systems is the loss (manipulation) of votes arising from transport between system components; for example, from interface to urn, and from urn to count module. This problem is exacerbated by changing the way in which votes are represented as they move through the system.

Fig. 7 Abstract ballot invariant axioms

```

axm18 : ∀b1 · b1 ∈ Ballots ⇒
        ((inv_1(b1) = TRUE) ⇔
         getNumberOfCandidates(b1) ∈
         MIN_NUMBER_OF_CANDIDATES .. MAX_NUMBER_OF_CANDIDATES)
        inv_1 requirement: the number of candidates is in range: MIN_NUMBER_OF_CANDIDATES
        .. MAX_NUMBER_OF_CANDIDATES
axm19 : ∀b1 · b1 ∈ Ballots ⇒
        ((inv_2(b1) = TRUE) ⇔
         getNumberOfPreferences(b1) ∈
         1 .. getNumberOfCandidates(b1))
        inv_2 requirement: the number of preferences is in range: 1 .. numberOfCandidates
axm20 : ∀b1 · b1 ∈ Ballots ⇒
        ((inv_3(b1) = TRUE) ⇔
         (∀c · c ∈ 1 .. getNumberOfCandidates(b1) ⇒
          getPreferenceForCandidate(b1 ↦ c) ∈ 0 .. getNumberOfPreferences(b1)))
        inv_3 requirement: only valid preferences in range 0 .. numberOfPreferences are allocated
        to each candidate
axm21 : ∀b1 · b1 ∈ Ballots ⇒
        ((inv_4(b1) = TRUE) ⇔
         (∀p · p ∈ 1 .. getNumberOfPreferences(b1) ⇒
          getCandidateForPreference(b1 ↦ p) ∈ 1 .. getNumberOfCandidates(b1)
         ))
        inv_4 requirement: only valid candidates in range 1 .. numberOfCandidates are allocated
        to each preference

```

In Fig. 8, we see how the way in which vote information is stored can change as votes move through the system.

In the interface, the voter conceptualises their vote as an array of candidates, some of whom are accorded preferences. However, in the ballot module, the count algorithm “sees” each vote as an ordered sequence of preferences. As votes are transformed from one representation to another, it is possible that a bug could transform a valid vote into an invalid vote [4].

The solution in Event-B is to model each of the ballots in the system as a machine, and to prove that the concrete machines are the refinements of the abstract machine. In Fig. 9, we see part of the Event-B model of the abstract ballot machine showing how the accessor `getNumberOfCandidates` is modelled. The other accessor methods are modelled similarly.

4.4 The tabulation process cannot change or invalidate ballots

This requirement is more complex than just stating that the number of ballots cannot change because the event that transfers a ballot may invalidate it without necessarily changing the ballot candidate-preference data. For example, if this event was to assign a value to the `currentCandidate`, which was out of range, then the transfer would break the ballot. Further, if the transfer was to assign a `currentCandidate` value that had already been previously rejected, then the ballot would also become invalid.

We are currently examining different modelling techniques for specifying and verifying such requirements that

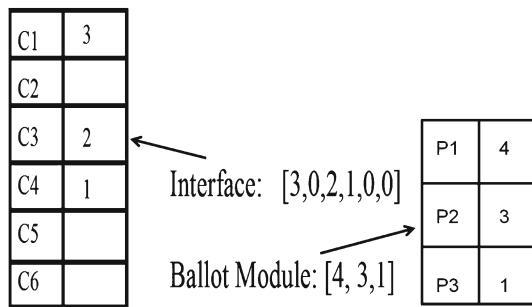


Fig. 8 Votes represented in different ways in the same system

```

MACHINE BallotAbstraction_mch
SEES BallotSpecification_ctx
VARIABLES
    ballot
INVARIANTS
    inv1 : ballot ∈ Ballots ∧
          invariant(ballot) = TRUE
EVENTS
Initialisation
    begin
        act1 : ballot :∈ ValidBallots
    end
Event getNumberOfCandidates ≐
    any
        result
    where
        grd1 : result = getNumberOfCandidates(ballot)
    then
        skip
    end
    
```

Fig. 9 Event-B specification of accessor getNumberOfCandidates

Fig. 10 BallotSpecification.java

```

package abstractions;
import tools.HasInvariant;

* Defines the abstract interface for ballots that are in the SAVE vote system
public interface BallotSpecification extends HasInvariant{

    * Each ballot must contain at least one candidate
    public final static int MIN_NUMBER_OF_CANDIDATES = 1;

    * Each ballot can have at most twenty candidates
    public final static int MAX_NUMBER_OF_CANDIDATES = 20;

    * Different ballots may have different numbers of candidates
    public int getNumberOfCandidates();

    * Different ballots may have different numbers of preferences
    public int getNumberOfPreferences();

    * @param preference for which we wish to find the corresponding candidate
    public int getCandidateForPreference(int preference) throws
    IllegalArgumentException;

    * @param candidate for whom we wish to find the associated preference
    public int getPreferenceForCandidate(int candidate) throws
    IllegalArgumentException;

    * The safety property that must be true for all valid instances :
    public boolean invariant();
}
    
```

seem to necessitate the addition of history variables that track not only the current state of an object, but also how it reached that state.

5 Critical requirements implemented in Java

Our development approach relies heavily on the coherent integration of the Event-B and Java code: our code is not generated from the specifications, rather the formal Event-B is used to document the code and to oblige the programmers to develop test code for invariant properties and functional requirements.

In Fig. 10, we see the Java implementation of the abstract ballot class.

In the cases where we have proven that our Event-B models are correct, this should be documented in the code. However, these proofs correspond only to the verification of design steps. They do not validate the natural language requirements and do not verify that the Java code is a correct implementation. Thus, we chose to complement our formal design process with rigorous unit testing. This helps in both the validation of the natural language requirements and in the verification of the Java implementation.

6 Critical analysis of lightweight integration

The main advantage offered by our lightweight integration is that we have a coherent set of documentations linking all our different development artefacts: natural language

requirements, formal specification, design steps, unit tests, and code. We found that we could easily manage changes to any of these artefacts, and the improved traceability speeded up our development process. Another important consequence of our OO approach was that early unit testing of abstract machine methods led to identification of errors in our interpretation of natural language requirements. Further, the reusable unit test code helped us to automatically find bugs in our implementation of the concrete classes.

The main disadvantage is that the Event-B OO style of specification does not fully leverage all advantages of OO modelling that arise out of concepts such as inheritance, polymorphism, dynamic binding, composition, aggregation etc. Our OO style could be enriched using a syntactic sugaring of the Event-B, but we chose instead to use the UML class diagrams to structure and annotate the Event-B with an OO interpretation.

7 More heavyweight alternatives

In this section, we consider four more heavyweight alternatives to the integration problem.

7.1 Using the OCL with the Event-B models

We originally considered using the OCL to narrow the gap between the Event-B and the UML. Our idea was to specify the natural language requirements (of invariants and pre/post conditions of methods) using the OCL and then to translate these to Event-B. This should help us to better structure the validation process: instead of trying to validate Event-B against natural language expressions, we would validate Event-B against the OCL models and then validate the OCL against the natural language. In our view, this approach—adding an extra bridge—could be beneficial if the development team includes experts in OCL, but our developers had more experience in Event-B than OCL, and therefore found no advantage of using the OCL to connect the other two modelling languages.

7.2 Using the OCL of the UML to replace Event-B

One could reasonably ask: why bother with the Event-B if the designers can model using the OCL? In our study, we found three main reasons why working only with the OCL would have hindered our development:

- The proof of the correctness of the ballot refinements initially generated proof obligations that could not be discharged automatically. Further investigation identified incompleteness in the natural language requirements that the unit testing had not revealed. The OCL could not have helped in finding this incompleteness.

- The equivalence of the different ballot implementations is easy to express in Event-B, but we did not find the same for modelling in the OCL. In other words, the richness of expression of Event-B is a major advantage.
- The RODIN tool [2] provides an excellent support for verification of well-definedness of complex predicates which has no equivalent for the OCL.

7.3 Moving from Event-B to JML and ESC/Java2 without the need to use UML

The integration of JML and ESC/Java2 [5] is an alternative bridging mechanism to the use of UML for the annotation of Event-B specifications with OO concepts. However, we argue that it is better to have an explicit (graphical) representation of the OO structure in the models (as in the UML) rather than one that is implicit in the code. Design is about making decisions and the UML models are the natural place in which these decisions can be explicitly documented.

7.4 Generating java directly from the Event-B

The final option is concerned with the automated generation of Java code from the Event-B models. In other words, the abstract requirements are incrementally refined until the Event-B is concrete enough for direct implementation. This is clearly the goal of *correctness-by-construction* methods, such as Event-B, but the current state-of-the-art certainly would not generate a high-quality OO code with respect to the meeting of non-functional (software engineering) requirements.

8 Conclusions

In this section, we summarise the main contributions of our research, identify important related work and propose future work.

8.1 Main contributions

The main weakness of our SAVE voting system is that the votes cannot be recounted by hand. The voter does have a paper record of their vote, but it is impossible for them to be decrypted and hand counted—the encrypted votes must be counted as a whole before the result is decrypted. The paper record of the vote allows a limited type of verifiability (or audit)—a voter can verify that their vote was counted after the election, but they cannot verify that it was correctly counted.

This encrypted-count mechanism has not been formally verified and so we need some means of checking that it is correct. Our approach uses the un-encrypted count (whose

development was rigorously driven by the Event-B models) as an oracle for testing the encrypted-count. The formal development design pattern that we followed is a generally applicable approach.

8.2 Related work

Recent work has shown how to use JML in the development of e-voting counting code [16]. The formal specification of test oracles has been reported over 15 years ago [15]. *Validating UML models and OCL constraints* [17] discusses the fundamental semantic issues in integrating OCL in UML. *Linking Event-B and Concurrent Object-Oriented Programs* [8] considers the modelling of communicating objects in Event-B. *Automated Validation of Class Invariants In C++ Applications, using UML design patterns* [10] is one of the first papers to consider a design pattern for automated validation of class invariants in an OO programming language. *Language and Tool Support for Class and State Machine Refinement in UML-B* [18] proposes an OO semantics for Event-B with emphasis on refinement. *Checking JML Specifications with B Machines* [3] suggests providing a formal semantics for JML models using B machines. An approach that uses a formal modelling language for generating Java unit tests was reported by Cheon and Leavens [7].

8.3 Future work

In the short-term, we plan to fully formalise the tabulation process using Event-B. In the medium-term, our lightweight integration approach will be evaluated on a wide range of case studies. In the long-term, our goal is a heavyweight, fully formal, integration of Event-B with UML. This will be based on a better understanding of the relationship between refinement and OO concepts.

References

1. Abrial JR (2010) Modeling in Event-B—System and Software Engineering. Cambridge University Press, Cambridge
2. Abrial JR, Butler M, Hallerstede S, Hoang TS, Mehta F, Voisin L (2010) Rodin: an open toolset for modelling and reasoning in Event-B. *Int J Softw Tools Technol Transf* 12:447–466
3. Bouquet F, Dadeau F, Gros Lambert J (2005) Checking JML specifications with B machines. In: Treharne H, King S, Henson MC, Schneider SA (eds) ZB, Lecture notes in computer science, vol 3455. Springer, Berlin Heidelberg, pp 434–453
4. Cansell D, Gibson JP, Méry D (2007) Refinement: a constructive approach to formal software design for a secure e-voting interface. *Electron Notes Theor Comput Sci* 183:39–55
5. Chalin P, Kiniry JR, Leavens GT, Poll E (2005) Beyond assertions: advanced specification and verification with JML and ESC/Java2. In: de Boer FS, Bonsangue MM, Graf S, de Roeper WP (eds) FMCO, Lecture notes in Computer Science, vol 4111. Springer, Berlin, pp 342–363
6. Chaum D (2004) Secret-ballot receipts: true voter-verifiable elections. In: *IEEE security & privacy* vol 2, No 1. IEEE, pp 38–47
7. Cheon Y, Leavens GT (2002) A simple and practical approach to unit testing: the JML and JUnit way. In: Proceedings of the 16th European conference on object-oriented programming, ECOOP'02. Springer-Verlag, London, UK, pp 231–255
8. Edmunds A, Butler M (2008) Linking Event-B and concurrent object-oriented programs. *Electron Notes Theor Comput Sci* 214:159–182
9. Fontaine C, Galand F (2007) A survey of homomorphic encryption for nonspecialists. *EURASIP J Inf Secur* 2007:1–15
10. Gibbs TH, Malloy BA, Power JF (2002) Automated validation of class invariants in C++ applications. In: Proceedings of the 17th IEEE international conference on automated software engineering, ASE'02. IEEE Computer Society, Washington, DC, USA, pp 205–215
11. Gibson JP (1993) Formal object oriented development of software systems using LOTOS. Thesis CSM-114, Stirling University
12. Gibson JP, Lallet E, Raffy JL (2008) Analysis of a distributed e-voting system architecture against quality of service requirements. In: The third international conference on software engineering advances (ICSEA 2008). IEEE Computer Society, pp 58–64
13. Gibson JP, Lallet E, Raffy JL (2009) Feature interactions in a software product line for e-voting. In: Feature Interactions in Software and Communication Systems X. IOS Press, pp 91–106
14. Gibson JP, Lallet E, Raffy JL (2010) Engineering a distributed e-voting system architecture: meeting critical requirements. In: Architecting Critical Systems, First International Symposium, ISARCS 2010, Prague, Czech Republic, June 23–25, 2010, Proceedings, vol 6150. Springer, pp 89–108
15. Hagar J, Bieman JM (1996) Using formal specifications as test oracles for system-critical software. *Ada Lett XVI*:55–72
16. Kiniry JR, Morkan AE, Cochran D, Oostdijk M, Hubbers E (2006) Formal techniques in a remote voting system. *SIGSOFT Softw Eng Notes* 31(6):1–2
17. Richters M, Gogolla M (2000) Validating UML models and OCL constraints. In: Evans A, Kent S, Selic B (eds) UML Lecture notes in computer science, vol 1939. Springer, Berlin, pp 265–277
18. Said MY, Butler MJ, Snook CF (2009) Language and tool support for class and state machine refinement in UML-B. In: Cavalcanti A, Dams D (eds) FM, Lecture notes in computer science, vol 5850. Springer, Berlin, pp 579–595