Safety Property Driven Test Generation from JML Specifications

Fabrice Bouquet, Frédéric Dadeau, Julien Groslambert, and Jacques Julliand

Université de Franche-Comté - LIFC - CNRS - INRIA
16 route de Gray - 25030 Besançon cedex France
Tel.: (+33)(0)381 666 664
email: {bouquet, dadeau, groslambert, julliand}@lifc.univ-fcomte.fr

Abstract. This paper describes the automated generation of test sequences derived from a JML specification and a safety property written in an ad hoc language, named JTPL. The functional JML model is animated to build the test sequences w.r.t. the safety properties, which represent the test targets. From these properties, we derive strategies that are used to guide the symbolic animation. Moreover, additional JML annotations reinforce the oracle in order to guarantee that the safety properties are not violated during the execution of the test suite. Finally, we illustrate this approach on an industrial JavaCard case study.

Keywords: automated testing, safety properties, black-box testing, Java Modeling Language, JavaCard.

1 Motivations

Annotation languages provide an interesting approach for the verification and validation of programs, allowing to describe, using annotations, the expected behavior of a class. Their advantage is to share a common level of abstraction with the considered programming language, which is useful in program verification/validation activities such as testing [11]. In this latter category, the Java Modeling Language [12] (JML) makes it possible to use lightweight annotations as well as heavyweight annotations to specify the behaviors of the methods. JML is well tool-supported and has shown is usefulness in industrial case studies, especially in the domain of JavaCard verification [6].

We propose an automated model-based testing approach for the validation of safety properties on a JavaCard application. A previous work [3], introducing JML-Testing-Tools[1] (JML-TT), has presented our ability to generate functional test sequences from a JML model, by performing the symbolic animation of the JML specification in order to reach a pertinent test target. We present in this paper the extension of this technology destined to the generation of test sequences that cover a user-defined safety property. This latter is expressed using the Java Temporal Pattern Language (JTPL) [19]. The JAG tool[2] [9] has been

[1] http://lifc.univ-fcomte.fr/~jmltt/
[2] http://lifc.univ-fcomte.fr/~groslambert/JAG/

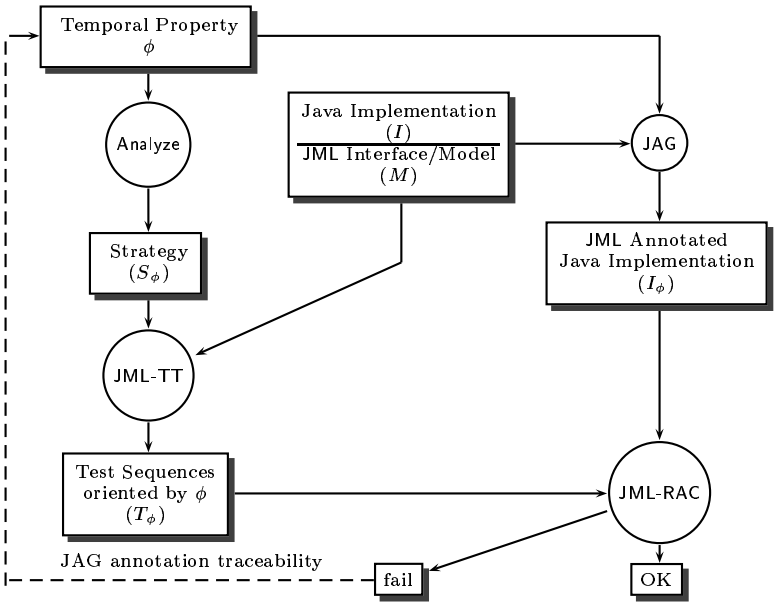

Fig. 1. General Approach

designed to use the JTPL in order to express and to check safety properties on a model or on an implementation, by generating extra JML annotations ensuring the satisfaction of the JTPL property. Our proposal is to combine JAG and JML-TT in order to generate test cases that are complementary of the functional ones and relevant w.r.t. the safety property.

For example, suppose that after the invocation of a method m, a property P must be established in all the states of the program. This property can be written with a JTPL pattern as follows:

$$\textbf{after } m \textbf{ called always } P \tag{1}$$

Then, only executions where the method m is invoked are relevant for this property. Therefore, we would like to generate only these kinds of executions.

Our approach is summarized in Fig. 1. First, we analyze the safety property ϕ and generate a strategy of test sequences generation S_ϕ. This strategy is built by combining test patterns according to the safety property schema. Second, from the JML interface M and S_ϕ, JML-TT computes a test suite, relevant w.r.t. ϕ and covering the functional behavior of the application. Then, these test sequences are executed on the annotated implementation I_ϕ generated by the JAG tool from the annotated implementation I enriched by annotations specifying the temporal property ϕ. These JML annotations provides the oracle that concludes on the verdict of the test. In addition, the extra annotations provide an oracle that concludes on the satisfaction of the property ϕ. If the annotations derived from this latter fail to be checked at run-time, and thanks to the JAG

traceability, we are able to retrieve the original temporal property that is not satisfied on the implementation, and furthermore, to retrieve the requirements of the security policies that have not been correctly implemented.

This paper is organized as follows. Section 2 presents the Java Modeling Language and its on-the-fly verification capacities, using the JML Runtime Assertion Checker (JML-RAC). Section 3 presents the JTPL temporal logic language, used to express the requirements of an application in terms of a temporal property. The generation of annotations for these properties is also described. Section 4 explains how the JML-TT framework computes test sequences driven by a safety property. Section 5 presents the result of an experiment made on a case study, and draws a comparison between our approach and a combinatorial test generation tool. Section 6 compares our approach with related works and discusses its originality. Finally, Section 7 concludes and presents the future work.

2 JML and Runtime Assertion Checking

The Java Modeling Language [12] is a behavioral interface specification language for Java programs, designed by G.T. Leavens et al. The specification consists in decorating a Java code or an interface in a comment-like syntax (`//@` for single-line annotations, `/*@ ...@*/` for multiple-line annotations). JML is based on the *Design By Contract* principles, stating that the system has to fulfill the methods requirements (i.e., their precondition) to invoke them. As a counterpart, the methods establish their postcondition.

JML considers different *clauses* to express the specifications. They involve the use of predicates in a Java-based syntax, enriched with specific JML keywords. Figure 2 presents an example of a JML specification. This specification describes a simplified electronic purse, specified by a balance (`bal`), to which money can be credited or withdrawn, using methods `init(byte,short)` and `complete()` to respectively initialize and complete the transaction. This specification illustrates the different clauses that can be used to design the JML model. The `invariant` clause describes the class invariant that applies on the class attributes. The method specifications are described using by specifying the precondition (`requires` clause), the normal postcondition (`ensures` clause) which gives the postcondition established when the method terminates normally, the exceptional postcondition (`signals` clause) which gives the postcondition that is established when the method throws an exception, and the list of the attributes which are modified by the invocation of the method (`assignable` clause).

The Runtime Assertion Checker is a compiler that enriches the Java bytecode with the checking of the different JML clauses. The execution of the RAC-compiled Java classes makes it possible to automatically check the specification predicates when running the program. If an execution violates one of the JML assertions, a specific exception is raised indicating which assertion has not been satisfied. Therefore, this feature is used as an oracle.

```
class Purse {

  //@ invariant max >= 0;
  protected short max;

  /*@ invariant bal >= 0 && bal <= max; */
  protected short bal;

  /*@ public normal_behavior
    @   assignable bal, max, transVal;
    @   ensures
    @        (m > 0  ==> max == m) &&
    @        (m <= 0 ==> max == 1) &&
    @        bal == 0 && transVal == 0;
    @*/
  public Purse(short m) {...}

  /*@ behavior
    @    requires   transVal != 0;
    @    assignable bal, transVal;
    @    ensures    bal ==
    @       (short) (\old(bal)+transVal);
    @    ensures    transVal == 0;
    @ also
    @    requires   transVal == 0;
    @    assignable \nothing;
    @    signals (IllegalUseException) true;
    @*/
  public void complete()
         throws IllegalUseException { ... }

  private short transVal;

  final static byte CREDIT_MODE = 0;
  final static byte DEBIT_MODE = 1;

  /*@ behavior
    @   requires a > 0 && transVal == 0;
    @   {|
    @     requires   P1 == CREDIT_MODE &&
    @           bal + a <= max;
    @     assignable transVal;
    @     ensures    transVal == a;
    @   also
    @     requires   P1 == DEBIT_MODE &&
    @           bal - a >= 0;
    @     assignable transVal;
    @     ensures    transVal == (short)(- a);
    @   |}
    @  also
    @    requires   (P1 != CREDIT_MODE &&
    @                P1 != DEBIT_MODE) ||
    @               a <= 0 || transVal != 0;
    @    assignable \nothing;
    @    signals (IllegalUseException) true;
    @*/
  public void init(byte P1, short a)
         throws IllegalUseException {...}

  ...  }
```

Fig. 2. Example of a JML specification

JML has two main uses, it can be used to reinforce the code and to help the proof of the code (e.g. using JACK [7]). Our philosophy is to consider JML as an entire specification language that does not require Java code to be employed. If the hypothesis may seem strong for all Java programs, we believe that it is worth doing the effort of writing a complete JML specification, with strong pre- and postconditions, in the domain of embedded programs, such as JavaCard [17]. In our approach, we use JML as a source for test target definition and model-based test cases computation. A recent evolution has been proposed to express temporal properties in JML, involving the use of the RAC. It is now described.

3 A Temporal Logic Extension for JML-like Language

We present an extension of JML with temporal specifications, first defined in [19]. This language, called *Java Temporal Pattern Language* (JTLP) is inspired by Dwyer's *specification patterns* [8]. Dwyer shows through a study of 500 specification examples, that 80 % of the temporal specification requirements can be covered by a finite number of formulae. This high-level temporal logic language for Java follows this philosophy, providing to the user structures to express common temporal requirements on Java classes. Moreover, the language can deal with both normal and exceptional terminations of methods. This language can be used to express safety or liveness properties. In this paper, we only focus on safety properties, for which we give the corresponding syntax and semantics. Readers can refer to [19] for a formal definition of the whole language semantics.

```
<Temp> ::= after <Events> <Temp>
         | before <Events> <TraceProp>
         | <TraceProp> unless <Events>
         | <TraceProp>

<TraceProp> ::= always <StateProp>
              | never <StateProp>
              | <TraceProp> and <TraceProp>
              | <TraceProp> or <TraceProp>
<StateProp> ::= <JMLProp>
              | <Method> enabled [ with <JMLProp>]
              | <Method> not enabled[ with <JMLProp>]
  <Events> ::= <Event>,<Events>
   <Event> ::= <Method> called [ with <JMLProp>]
             | <Method> normal [ with <JMLProp>]
             | <Method> exceptional [ with <JMLProp>]
             | <Method> terminates [with <JMLProp> ]
```

Fig. 3. Syntax of the safety patterns

3.1 Syntax and Semantics of the Language

The syntax of the subset of the JTPL language expressing safety properties is displayed given in Fig. 3. This language is based of the notions of *events* and *state properties*.

Events can be either: (*i*) m **called**, meaning that the method m has been called, without considering the method terminasion; (*ii*) m **normal**, meaning that the method m has terminated normally; (*iii*) m **exceptional**, meaning that the method m has terminated exceptionally (by throwing an exception); (*iv*) m **terminates**, meaning that the method m has terminated (either normally or by throwing an exception). The events can be enriched with a predicate P introduced by the keyword **with**. Thus, m **called with** P means that m has been called within a state satisfying P and m **terminates** (resp. **normal** and **exceptional**) **with** P means that m terminates (resp. terminates normally and terminates by throwing an exception) in a state satisfying the predicate P.

A state property P can be either: (*i*) a JML predicate; (*ii*) m **enabled**, meaning that if the method m is called and if the method m terminates, then it terminates normally; (*iii*) m **not enabled**, meaning that if the method m is called and if the method m terminates, then it terminates exceptionally, i.e., by throwing an exception.

The state properties m **enabled** and m **not enabled** are especially designed to express properties on JavaCard applets, since JavaCard commands can be called from any state. Thus, once a method is called, either the call is licit w.r.t. the expected state variable values and the parameters values and thus the method terminates normally, or the call is illicit and the method terminates exceptionally. Notice that these two state properties are true if the method is not called or if the method is called but does not terminate (i.e., the method diverges). This clause can also be enriched with a predicate P introduced by the keyword **with**.

Finally, events and state properties can be combined with the keywords of the language: (*i*) **always** P, which is true on an execution σ if the state properties

```
class Purse {

  //@ ghost boolean inProgress = false;
  ...

  /*@ behavior
    @    ...
    @    ensures    inProgress == false;
    @ also
    @    ...
    @*/
  public void complete()
                throws IllegalUseException {
     //@ set inProgress = false;
     ...
  }
```

```
  /*@ behavior
    @    ...
    @    ensures inProgress == true;
    @    ensures \old(inProgress) ==> false;
    @ also
    @    ...
    @*/
  public void init(byte P1, short a)
                throws IllegalUseException {
       ...
     }
     finally {
        //@ set inProgress = true;
     }
  }
}
```

Fig. 4. Example of annotations produced by the JAG tool.

P holds on every state of σ; (ii) **never** P, which is true on an execution σ if the state properties P never holds on any state of σ. It is equivalent to **always** $\neg P$; (iii) C **unless** E, which is true on an execution σ if the trace property C is satisfied on the segment of σ ending with an event in E, or if the trace property C is satisfied on the whole of σ and no event in E happens; (iv) **before** E C, which is true on an execution σ if any occurrences of an event in E is preceded by a prefix of σ satisfying the trace property C; or (v) **after** E T, which is true on an execution σ if the suffix of σ starting with any event in E satisfies the temporal formula T. Notice that conjunctions and disjunctions, respectively denoted by **and** and **or**, have a standard meaning.

This specification language is an input of the JAG tool, presented in the next subsection.

3.2 Translation of JTPL into JML with JAG

The JAG tool [9] generates JML annotations ensuring a given temporal property. As an illustration, we present a safety property that has to hold on the example of Fig. 2, specifying that after a successful `init`, one can invoke `init` once again only if the transaction has been validated by invoking `complete`:

after `init` **normal**
always `init` **not enabled unless** `complete` **called** (S_0)

The additional annotations, automatically generated and related to this property, are given in Fig. 4. This property is expressed by:

- a *ghost* boolean variable `inProgress`, initialized to `false`. This variable is set to `true` when the event `init` **normal** occurs and set to `false` again when `complete` **called** occurs.
- a postcondition ensuring that `init` cannot terminate normally when variable `inProgress` is equal to `true`. This predicate reinforces the normal postcondition by preventing it from being evaluated to true if `inProgress` is false, stating, as a consequence, that the method can not terminate normally in this particular case.

The interested reader will find in [19] the details of the translation, for all structures of the language.

4 Test Generation from Temporal Properties

We describe in this section the definition of the principles which consist in animating the specification according to a given temporal logic property. Then, we present the coverage criteria that we apply on the specification. Finally, we explain the test sequence computation.

4.1 Principles

Our approach is an extension of the previous work about functional test generation that is presented in [4] on the symbolic animation of JML specifications. The principle of our approach is to associate a test suite to each safety property we consider. Thus, we perform the symbolic animation of the specification in order to build a test sequence that exercises the property, by activating the behaviors of the JML specification.

The computation of the test sequences is driven by a strategy derived from the temporal formula, to guide the animation of the specification. A strategy is composed of a sequence of steps in which our aim is to activate a particular behavior of the specification or to cover all the behaviors. When the last step is done, the test generation stops. In addition, we consider a bound that limits the test sequences length, and guarantees the termination for each step of the research.

In addition, we rely on the JML annotations describing the safety property, and produced by the JAG tool, to complete the oracle. Thus, if one of these annotations fails to be checked at run-time, we are able to provide to the user an indication concerning the original temporal property and the original requirement that have been violated.

4.2 Coverage Criteria

Our approach considers the classical coverage of the specification, at three levels: the specification coverage, the decision coverage, and the data coverage.

Specification coverage The specification coverage is achieved by activating the different behaviors that we extract from the JML method specifications. Figure 5 describes the extraction of behaviors from a JML method specification. A behavior is represented by a path leading from node 1 to node 0. According to this figure, we assume that the method may deterministically terminate (expressed by T) either normally ($T =$ `no_exception`) or by throwing one of its M specified exceptions ($T = E_i$ for $1 \leq i \leq M$). We also assume that the exceptional behaviors are deterministic, which means that their guards are mutually exclusive. *De facto*, the behaviors of the methods only depend on the current state variables values, and the parameter values.

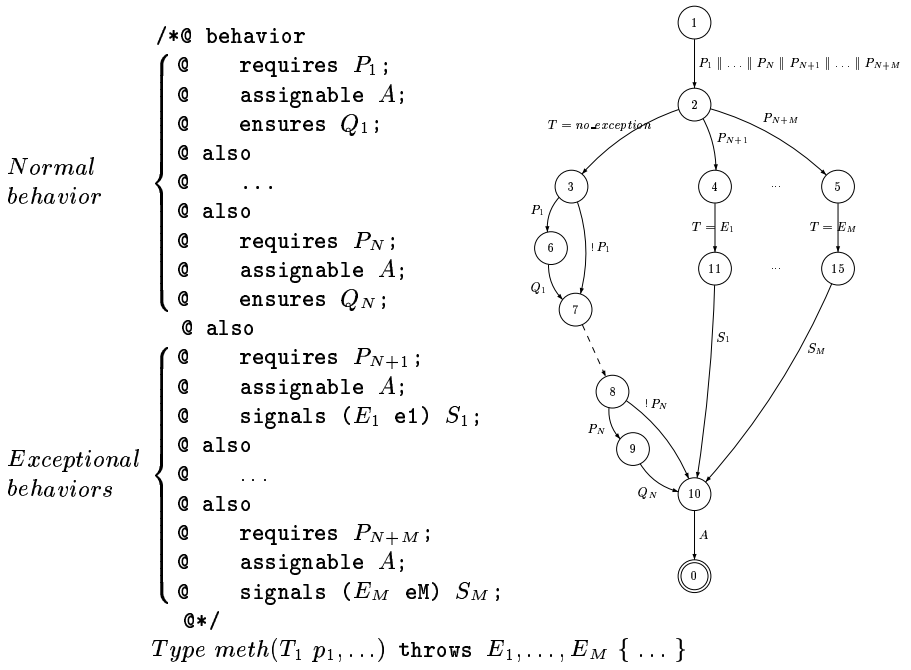

Fig. 5. Extraction of the behaviors from a JML method specification

Decision coverage We achieve the decision coverage by rewriting the disjunctions within the preconditions of the JML specifications. We consider four rewritings of the disjunctions, described by $a \vee b$. Rewriting 1 consists in leaving the disjunction unmodified. Rewriting 2 consists in creating a choice between the two predicates (a [] b). Thus, the first branch and the second branch independently have to succeed when being evaluated. Rewriting 3 consists in creating an exclusive choice between the two predicates ($a \wedge \neg b$ [] $\neg a \wedge b$). Only one of the sub-predicates of the disjunction is checked at one time. Rewriting 4 consists in testing all the possible values for the two sub-predicates to satisfy the disjunction ($a \wedge \neg b$ [] $\neg a \wedge b$ [] $a \wedge b$). Each one of these rewritings guarantees at least one decision coverage.

Data coverage When performing the symbolic animation of the specification, the input parameters of the methods that are invoked are left unspecified, and their symbolic values are managed by constraint solvers. When the symbolic sequence computation is over, we select the boundary values for the unspecified parameters. More details about the application of this work to JML can be found in [3].

In addition to these classical coverage criteria, we are especially interested in exercising the temporal property. This is achieved by defining different strategies that are in charge of activating the JML method behaviors w.r.t. the temporal property. We now describe these strategies, which represent the main contribution of the paper.

Strategy(**after** Events Temp) = Research(Events); Strategy(Temp)
Strategy(**before** Events TraceProp) = Cover; Strategy(TraceProp); Research(Event)
Strategy(TraceProp **unless** Events) = CoverStop(Events); Strategy(TraceProp)
Strategy(**always** StateProp) = Cover;Strategy(StateProp)
Strategy(StateProp_1 **and** StateProp_2) = Strategy(StateProp_1) [] Strategy(StateProp_2)
Strategy(StateProp_1 **or** StateProp_2) = Strategy(StateProp_1) [] Strategy(StateProp_2)
Strategy(<JMLProp>) = ϵ
Strategy(m **enabled** [**with** <JMLProp>]) = Active(m **exceptional** [**with** <JMLProp>])
Strategy(m **not enabled** [**with** <JMLProp>]) = Active(m **normal** [**with** <JMLProp>])

Fig. 6. Strategies for the JTPL language

4.3 Test Sequence Computation

The test sequence computation strategy depends on the safety property that has been defined. According to the pattern that matches the temporal property, a specific strategy is employed.

The translation from JTPL into JML-TT strategies is described by the function Strategy given in Fig. 6, in which ϵ denotes that no strategy is applied. A strategy consist of sequences (denoted by ";") or choices (denoted by "[]") of steps among the four following patterns:

- **Research of** E (Research(E)). This strategy performs a *best-first* algorithm that aims at activating an event in E. This principle has been already described in [3].
- **Coverage of behaviors** (Cover). This strategy performs the symbolic animation of the specification in order to cover all the behaviors. This is done by a *depth-first* algorithm that activates the normal behaviors of the model. The main advantage of using the specification is that it delays the combinatorial explosion occurring during the exploration of the reachability graph by filtering the sequence of methods, so as to comply with the methods contracts. When a behavior is newly activated, the current execution sequence is returned to provide a test case. A "backtracking" mechanism makes it possible to resume the depth-first research.
- **Coverage of behaviors with stop on** E (CoverStop(E)). This strategy is similar to the previous one, but the depth-first algorithm stops when an event in E is activated. As in the previous case, a backtracking occurs to resume the computation.
- **Activation of** E (Active(E)). This consists in a systematic activation of the events in E. This step is crucial since it will be used to activate the expected or unexpected events, expressed in the property. For example, if the state property is of type m [**not**] **enabled**, the method m is tried to be activated. This step is also performed using a depth-first algorithm. (Un)Expected behaviors are detected by considering the specification and performed in a try-catch mecanism that is in charge of either catching a unexpected exception, or throwing a specific exception when the expected exceptional behavior has not been thrown.

Notice that the sequences Cover;CoverStop(E) and CoverStop(E);Cover are reduced to CoverStop(E).

Example 1 (Strategy for S_0). The strategy associated to S_0, given by the function Strategy of Fig. 6 is the following:

Research(`init` **normal**); CoverStop(`complete` **called**); Active(`init` **normal**)

It corresponds to the following steps: (*i*) we research a sequence that ends with the activation of the normal behavior of `init`; (*ii*) we cover all the behaviors of the class, the research is stopped when the event `complete` **called** occurs; (*iii*) we try to activate the normal behavior of method `init` to test if `init` is effectively **not enabled**.

This automatic test generation approach, using the strategies explained above, has been applied to a case study. The results of this experiment are exposed in the next Section.

5 Experiment of a Case Study

We now present an experiment that we have done on a case study. We start by describing the specification, before expressing the temporal properties from which we want to generate test cases, and finally we compare our approach with a similar tool.

5.1 Presentation of the Demoney Specification

Demoney is an electronic purse developed by Trusted Logic [15]. This JavaCard application makes it possible to pay a purchase in a store using a terminal and can be credited from cash or from a bank account in an ATM. Demoney is not an industrial application but is complex enough to handle typical features and security problems related to banking systems.

Similarly to the other JavaCard applications, the life cycle of the card starts with a personalization phase, where particular variables, such as maximum balance amount, are fixed using the `PUT_DATA` command. Then, a `STORE_DATA` command stores the personalization variables. The application can only be personalized once. There are four access levels (public, debit, credit and admin), which restrict the activation of the commands. For example, the `STORE_DATA` command can only be invoked with the admin access level. Access levels can be selected using the `INITIALIZE_UPDATE` and `EXTERNAL_AUTHENTICATE` commands. For a successful change, the methods have to be atomically invoked, e.g. `INITIALIZE_UPDATE` must immediately be followed by `EXTERNAL_AUTHENTICATE`. `INITIALIZE_TRANSACTION` and `COMPLETE_TRANSACTION` are used to perform transactions, whose types (debit or credit from cash or from bank) are expressed using parameter `P1` of the first command. These two commands also have to be atomically invoked for a successful transaction. For a credit from

a bank account, the PIN code of the card must have been checked using the VERIFY_PIN command. The number of tries is limited and chosen at the personalization time. Finally, when the pin is blocked after unsuccessful VERIFY_PIN invocations, it is possible to unblock the card using the PIN_CHANGE_UNBLOCK command.

For the test generation, we use a JML model of Demoney designed from the informal public specification. This model represents over 500 lines of JML and has been validated with the JML-TT Symbolic Animator [4].

5.2 Temporal Properties

We illustrate the test generation on two safety properties. In addition, in order to pilot the test generation and to have interesting test sequences, we add some requirements on the state in which the considered commands terminate. These requirements are used to force the first part of the test cases to configure the card with interesting values for the maximal balance on the card (maxBalance), the maximal debit amount (maxDebit) and the pin code (pin.code). These requirements are expressed using the **with** clause of the JTPL expressions, by a context predicate C:

```
maxBalance == 10000 & maxDebit == 5000 & pin.code == 1234
```

We address the verification of the two following safety properties.

1. The personalization is unique:

after STORE_DATA **normal with** C **always** STORE_DATA **not enabled** (S_1)

2. When the pin is blocked, it is impossible to credit the card from a bank unless a successful call to the PIN_CHANGE_UNBLOCK method in the unblocking mode (expressed by value UNBLOCK for parameter P1).

after VERIFY_PIN **terminates with** pin.tries == 0 & C
always INITIALIZE_TRANSACTION **not enabled**
with P1 == CREDIT_FROM_BANK
unless PIN_CHANGE_UNBLOCK **normal with** P1 == UNBLOCK; (S_2)

Using the JAG tool, we generate the JML annotations that ensures the satisfaction of these properties. The challenge is to validate the implementation w.r.t. these temporal properties. According to Sect. 4.3, the JAG tool computes the following strategies for S_1 and S_2:

Research(STORE_DATA **normal with** C); Cover; Active(STORE_DATA **normal**) (S_1)

Research(VERIFY_PIN **terminates with** pin.tries == 0 & C);
CoverStop(PIN_CHANGE_UNBLOCK **normal with** p1 == UNBLOCK);
Active(INITIALIZE_TRANSACTION **with** P1 == CREDIT_FROM_BANK) (S_2)

These strategies are used in JML-TT to drive the automated test generation previously explained. Results of the generation for these two properties are now presented, and a comparison with a combinatorial test generation tool is exposed.

5.3 Results, Comparison, and Discussion

Tests have been generated for different values n of the depth search. An example of test generated for the property S_1 is displayed in Fig. 7. The test is composed as follows: (a) a *prelude* reaches a state where `STORE_DATA` is activated under the C condition; (b) we try to cover a particular behavior (here `COMPLETE_TRANSACTION`); (c) the method `STORE_DATA` is activated once again; (d) a `try...catch` statement observes if a exception has been thrown by the execution of `STORE_DATA`. Table 2 displays the general results of the test generation. We remark that the number of test cases is twice the number of behaviors covered. This is explained by the boundary values selection which both minimizes and maximizes data values. For each property we cover all the reachable behaviors (13 for S_1, 8 for S_2) for a reasonable depth, with a minimal number of test cases.

The test suites driven by temporal properties complement the test suites we obtained using the functional techniques presented in [3], as it was expected. Moreover, these two approaches do not generate the same test cases, since the functional test cases try to activate each behavior by reaching the shortest path leading to a state that makes it possible to activate it. For example, the test case, displayed in Fig. 7, is not produced in the functional approach. Since these two approaches can not be compared, i.e., they do not aim at the same purpose, we wanted to draw a comparison with a tool that has a similar approach to ours: Tobias [13].

Tobias is a combinatorial test generation tool that uses user-defined regular expressions to build test sequences consisting of Java method calls. This approach then relies on the JML-RAC to provide an oracle that gives the test verdict. Since this approach does not consider the JML specification for the generation, it may produce *inconclusive* tests, when the precondition of a method is not satisfied. Both Tobias and our approach consist in semi-automatic testing, since a user is asked to respectively provide a test schema or a safety property.

Safety Property	n	# of tests	# of behaviors covered
S_1	1	10	4/17
S_1	2	12	5/17
S_1	3	18	9/17
S_1	4, 5, 6	22	11/17
S_1	7	24	12/17
S_1	≥ 8	26	13/17
S_2	1	8	4/17
S_2	2	12	6/17
S_2	3, 4	14	7/17
S_2	≥ 5	16	8/17

Table 2. Results of experiments with JML-TT

We tried to cover the property S_1 using the following Tobias test schema:

$$\texttt{prelude}; \texttt{STORE\_DATA}; (\texttt{other\_methods})^{0..n}; \texttt{STORE\_DATA}$$

This test schema's automaton recognizes the test cases that we have produced. With $n = 4$, this schema produces roughly 45436 test cases, of which 10% are relevant, and covering 11 behaviors, as we also do. The `prelude` part consists in configuring the card before the personalization. This part had to be manually generated.

The second property gave similar results, asking much more effort to manually define the `prelude` and describe the remainder of the test schema.

This experiment shown the advantages of our approach, since: (*i*) we achieved a higher level of automation both in the test case generation; (*ii*) we mastered the combinatorial explosion and created less test cases which are all relevant (since they are based on the symbolic animation of the model), and which cover all the reachable behaviors for a given depth; (*iii*) the effort asked to the user is minimal and requires less expertise from the validation engineer, since he only has to describe a temporal property (and its optional context) instead of providing subsets of the test sequences; (*iv*) the expressiveness of our approach, and especially the possibility of expressing an optional context, allows to subtly drive the test generation.

```
Demoney v = new Demoney();                                    ↑
v.INITIALIZE_UPDATE((byte) 3,(byte) 1);                       |
v.EXTERNAL_AUTHENTICATE((byte) 11,(byte) 0);                  |
v.PUT_DATA((byte) 3,(byte) 15,(short) 1234);                  | a
v.PUT_DATA((byte) 2,(byte) 0,(short) 5000);                   |
v.PUT_DATA((byte) 1,(byte) 0,(short) 20000);                  |
v.STORE_DATA((byte) 80,(byte) 0);                             ↓
v.INITIALIZE_UPDATE((byte) 2,(byte) 1);                       ↑
v.EXTERNAL_AUTHENTICATE((byte) 1,(byte) 0);                   |
v.INITIALIZE_TRANSACTION((byte) 1,(byte) 0,                   | b
                         (short) 20000);                      |
v.COMPLETE_TRANSACTION((byte) 0,(byte) 0);                    ↓
v.INITIALIZE_UPDATE((byte) 3,(byte) 1);                       ↑
v.EXTERNAL_AUTHENTICATE((byte) 1,(byte) 0);                   | c
try {                                                         |
  v.STORE_DATA((byte) 80,(byte) 0);                           ↓
  throw new JMLTTUnraisedException                            ↑
     ("IllegalUseException");                                 |
}                                                             | d
catch (IllegalUseException e) {                               |
    // Nothing to do in this case.                            |
}                                                             ↓
```

Fig. 7. A test case generated by JML-TT

6 Related Work

Testing Java programs using JML annotations has already been well studied and other tools are available. Korat [5] aims at providing an exhaustive set of structures satisfying a Java predicate, using SAT solving technologies. This approach has been adapted to JML, and relies on the method preconditions to build satisfying test data. Whereas Korat only considers an object creation and a method invocation, our approach proposes to build complex test sequences of method invocations. Jartege [16] produces stochastic test cases based on a Java program. The Runtime Assertion Checker is used when the test sequence is being built, in order to filter the irrelevant method invocations. The major advantage of Jartege is its full automation, but its main problem is the absence of startegy in the test generation which prevents it from being used in the domain of JavaCard.

Most of LTL based testing approaches use model-checkers such as Spin [10] to generate test cases. By fully exploring the state space of a model of the application, a model checker verifies that every configuration of the model satisfies a given property. When the property is not satisfied, the model checker exhibits a counter-example, i.e., a run of the model that does not satisfy the property. Approaches based on model checking use this counter-example mechanism to produce traces that can be transformed in test sequences. Sokolsky and al. [18], for a given LTL formula ϕ, compute a set of $\exists$LTL formulae that covers every subformulae of ϕ. In [2], Ammann and al. propose the mutation of the model or of the property to generate the counter-examples and then, the test suite. Both approaches need to use a finite abstraction of the model to generate the tests.

Our approach, based on symbolic animation and constraint solving that reduces the state place explosion, can handle potentially huge or infinite models. Although the JML-TT framework does not provide a complete exploration of the state space, it shows its effectiveness in practice.

The coverage metrics of the temporal property is an important and well-studied criteria for selecting relevant test cases. The approach of Sokolsky in [18], relies on the concept of *non-vacuity* in model-checking, capturing traces of the model that satisfy a property non-trivially. Implicitly, we have use this notion on our approach, since for each pattern of the language, we only generates tests relevant for the property.

7 Conclusion and Future Work

In this paper, we proposed an extension of the JML-TT framework, for the generation of test suites driven by safety properties. Based on the experimentation on a case study, this approach has shown its complementarity with the existing techniques of test for Java/JML and has led to effective results.

Our next task is to establish the coverage of the test suites in terms of coverage criteria of the safety property. Intuitively, it requires to consider the Büchi automaton extracted from the property and to define coverage in terms of states, transitions, or paths.

Our approach can be easily adapted to other specification languages such as SPEC$\sharp$ [14] or B [1]. One of the future challenge is to generalize the methodology presented in this paper to other temporal specification languages supported by the JAG tool. In particular, we are interested in LTL.Model-checking techniques, such as presented in Section 6, based on mutation of the formula or the model, can also be adapted to our automatic test generation framework, since our approach is close to bounded model-checking.

Another interesting future work is using property-driven generation for a collaboration between proof and test techniques. Using the JAG tools, one can generate the JML annotations on the implementation of an application and trying to prove it with proof obligation generator such as Jack [7]. If the proof of a generated annotation fails, and using the JAG traceability, we are able to retrieve

the temporal annotations and generate, via JML-TT, intensive test sets related to this particular property.

References

1. J.-R. Abrial. *The B Book*. Cambridge University Press, 1996.
2. P. Ammann, P. E. Black, and W. Majurski. Using model checking to generate tests from specifications. In *ICFEM'98*, pages 46–55. IEEE Comp. Soc. Press, 1998.
3. F. Bouquet, F. Dadeau, and B. Legeard. Automated Boundary Test Generation from JML Specifications. In *FM'06*, LNCS. Springer, 2006. To appear.
4. F. Bouquet, F. Dadeau, B. Legeard, and M. Utting. Symbolic Animation of JML Specifications. In *FM'05*, LNCS 3582, pages 75–90. Springer-Verlag, 2005.
5. C. Boyapati, S. Khurshid, and D. Marinov. Korat: automated testing based on java predicates. In *ISSTA'02*, pages 123–133. ACM Press, 2002.
6. C-B. Breunesse, N. Cataño, M. Huisman, and B. Jacobs. Formal methods for smart cards: an experience report. *Sci. Comput. Program.*, 55(1-3):53–80, 2005.
7. L. Burdy, A. Requet, and J.-L. Lanet. Java Applet Correctness: a Developer-Oriented Approach. In *FME'03*, lncs 2805, pages 422–439. Springer, 2003.
8. M. B. Dwyer, G. S. Avrunin, and J. C. Corbett. Patterns in property specifications for finite-state verification. In *ICSE*, pages 411–420, 1999.
9. A. Giorgetti and J. Groslambert. Jag: Jml annotation generation for verifying temporal properties. In L. Baresi and R. Heckel, editors, *FASE*, volume 3922 of *LNCS*, pages 373–376. Springer, 2006.
10. G.J. Holzmann. The model checker SPIN. In *IEEE Trans. on Software Engineering*, volume 23-5, pages 279–295, 1997.
11. G. T. Leavens, Y. Cheon, C. Clifton, C. Ruby, and D. R. Cok. How the Design of JML Accomodates Both Runtime Assertion Checking and Formal Verification. In *FMCO*, volume 2852 of *LNCS*, pages 262–284. Springer, 2002.
12. G.T. Leavens, A.L. Baker, and C Ruby. JML: A notation for detailed design. In H. Kilov, B. Rumpe, and I. Simmonds, editors, *Behavioral Specifications of Businesses and Systems*, pages 175–188. Kluwer Academic Publishers, Boston, 1999.
13. Y. Ledru, L. du Bousquet, O. Maury, and P. Bontron. Filtering tobias combinatorial test suites. In M. Wermelinger and T. Margaria, editors, *FASE 2004*, volume 2984 of *LNCS 2984*, pages 281–294. Springer-Verlag, 2004.
14. K.R.M. Leino M. Barnett and W. Schulte. The Spec# Programming System: An Overview. In *Proceedings of the International Workshop CASSIS'04*, LNCS 3362, pages 49–69. Springer-Verlag, 2004.
15. R. Marlet and C. Mesnil. Demoney: A demonstrative electronic purse - card specification. Technical report, Trusted Logic, 2002.
16. C. Oriat. Jartege: A tool for random generation of unit tests for java classes. In *SOQUA 2005*, volume 3712 of *LNCS*, pages 242–256. Springer-Verlag, 2005.
17. Sun microsystems. *Java Card 2.1.1 Virtual Machine Specification*, May 2000. http://java.sun.com/products/javacard/javacard21.html#specification.
18. L. Tan, O. Sokolsky, and I. Lee. Specification-based testing with linear temporal logic. In *IRI*, pages 493–498. IEEE Systems, Man, and Cybernetics Society, 2004.
19. K. Trentelman and M. Huisman. Extending JML Specifications with Temporal Logic. In *AMAST'02*, number 2422 in LNCS, pages 334–348. Springer, 2002.