JML-based Verification of Liveness Properties on a Class in Isolation*

Julien Groslambert
Université de Franche-Comté - LIFC - CNRS
16 route de Gray
25030 Besançon cedex
France
groslambert@lifc.univ-fcomte.fr

Jacques Julliand
Université de Franche-Comté - LIFC - CNRS
16 route de Gray
25030 Besançon cedex
France
julliand@lifc.univ-fcomte.fr

Olga Kouchnarenko
Université de Franche-Comté - LIFC - CNRS - INRIA CASSIS
16 route de Gray
25030 Besançon cedex
France
kouchna@lifc.univ-fcomte.fr

ABSTRACT

This paper proposes a way to verify temporal properties of a Java class in an extension of JML (Java Modeling Language) called JTPL (Java Temporal Pattern Language). We particularly address the verification of *liveness* properties by automatically translating the temporal properties into JML annotations for this class. This automatic translation is implemented in a tool called JAG (JML Annotation Generator). Correctness of the generated annotations ensures that the temporal property is established for the executions of the class in isolation.

1. INTRODUCTION

Recently, significant progresses have been made in the field of smart card application verifications. The development of the Java Modeling Language project[1] (JML) is a part in these results [6]. The JML project defines a specification language which is syntactically and semantically close to Java, thus making specifications more accessible to Java programmers. JML allows adding to the Java class traditional formal annotations like method pre- and postconditions and class invariants. However, it is difficult to directly specify complex dynamic properties in JML, like temporal properties [15] that are often needed to express the security policies that the Java implementation has to ensure. For example, no JML clause permits easy expression of the followings properties

$$\begin{array}{c} \textit{“After the invocation of the method } \texttt{m} \\ \textit{the property } P \textit{ must be satisfied in all the states”}. \qquad (S) \\ \textit{“After the non-exceptional termination of the method } \texttt{m} \\ \textit{a state where } P \textit{ holds must inevitably} \\ \textit{be reached in the future”}. \qquad (L) \end{array}$$

Following [15], Property S is a safety property, expressing that *something bad* – a state where $\neg P$ holds after the invocation of `m` – must never happen whereas L is a liveness property, expressing that, under certain conditions – an invocation of `m` –, *something good* – a state where P holds – must *inevitably* happen in the future.

A key concept for reasoning modularly about safety properties is the notion of *class invariant*, i.e., a predicate over the variables of the class that must hold along all the history of all the instances of the class. Reasoning in a modular way about invariant consists in: (a) considering only the class *in isolation* [1], i.e., without regarding the program using the class, we show that the constructor of the class establishes the invariant and that each method of the class preserves it. (b) showing that under certain conditions on the program, the invariant is still satisfied.

We propose to address the expression and the verification of liveness properties following the same approach. Therefore, this paper particularly focuses on the first task, i.e., the verification of *liveness* properties in isolation.

For that, we provide in the paper (1) an execution path semantics of a Java Class in isolation and a semantics of the main JML annotations; (2) a primitive liveness operator Loop, inspired from [9], for expressing liveness properties; (3) a way to verify on a class in isolation liveness properties expressed with the Loop primitive, by generating JML annotations ensuring their satisfaction; (4) a tool, called JAG [10] (for JML Annotation Generator) implementing an automatic translation of liveness properties into standard JML annotations that ensure the satisfaction of these properties. Translation of liveness properties is done through the primitive Loop operator.

We particularly focus on Java card applet. So, this paper focus on single Java Class in isolation and the problem of inheritance and sub-typing is not addressed.

The paper is outlined as follows. Section 2 quickly presents JML on an example. Section 3 presents the semantics frame-

*Research partially funded by French Research ACI *Geccoo*.
[1]See `http://www.jmlspecs.org`.

Fifth International Workshop on Specification and Verification of Component-Based Systems (SAVCBS 2006) November 10-11

work of the paper. In particular, Section 3.1 defines a semantics for the class C in isolation whereas Section 3.2 gives the semantics of JML main annotations. Section 4 presents the verification of liveness properties on a class in isolation, through appropriate annotation generation. Section 5 presents the JAG tool, implementing this automatic generation of annotations. Section 6 concludes and presents the perspectives of future work.

Nota Bene: The proofs of propositions and theorems are not included in this paper, but can be found in [14].

2. OVERVIEW OF JML AND EXAMPLE

JML (Java Modeling Language) [16] is a specification language especially tailored for Java applications. Originally, JML was proposed by G.T. Leavens and his team; the development of JML is now a community effort. JML has been successfully used in several case studies to specify Java applications, and notably to specify smart card applications, written in Java Card [6, 13]. JML is developed following the Design by Contract approach [21], where classes are annotated with class invariants and method pre- and post-conditions. The predicates are written as (side-effect-free) boolean Java expressions, extended with specific constructs. Specifications are written as Java comments marked with a `@`, i.e., annotations follow `//@` or are enclosed between `/*@` and `@*/`. Below, we give a brief introduction to the main specification constructs of JML, by means of the example in Fig.1.

The class `Buffer` works as follows. A method `storeData()` personalizes the application by setting the length of the transaction. One can initialize a new transaction with the method `begin()`, creating a new temporary `buffer`. Then, a `write()` method fills the modifications in the temporary `buffer`, that is validated, i.e., assigned to the attribute `status`, by an invocation of `commit`. It is also possible to abort the transaction (`abort()`).

Figure 1 presents the main JML annotations on the simple example of a buffer. It shows a declaration of a class `invariant`, denoting a predicate that has to hold before and after every method call, i.e., in so-called JML *visible* states. History `constraints` allow expressing a relation between the pre- and post-state of all methods. Pre-state values of expressions are denoted by the JML keyword `\old`. Using the clause `for`, one may specify the list of the methods for which the history constraint must be satisfied. When this clause is omitted, the constraint must hold for all the methods of the class. The clause `requires` denotes the precondition of the method, i.e., a predicate that must be true when the method is called. A postcondition is expressed with an `ensures` clause. A method may terminate exceptionally by throwing an exception and satisfying the exceptional postcondition (`signals` clause). The `assignable` clause gives the list of variables that can be potentially modified by the method. A method without side-effect is denoted by the keyword `pure`. The specification of a method can also contain a `diverges` clause (not displayed in this example). If the predicate of a `diverges` clause of a method m is satisfied by the pre-state of m, then the execution of m *may* not terminate. Otherwise the method *must* terminate. By default, the JML `diverges` clause is set to `false`. A method with a `helper` modifier *may* not preserve the invariant. JML also introduces its own variables – declared with the keyword `ghost`. A special `set` annotation exists to assign their value.

In the rest of the paper, given a method m, we denote by `requires`(m) (resp. `diverges`(m)), the predicate of the `requires` clause of m (resp. the `diverges` clause). The correctness of a Java class C w.r.t. a JML annotation $\mathcal{A}$, denoted $C : \mathcal{A}$ in the rest of the paper, can be established by model-checking [24] or by a prover (B or Coq) via a proof obligation generator (Jack[8] or Krakatoa [20]).

3. A JAVA EXECUTION SEMANTICS

Linear temporal properties [23] have a semantics over infinite executions of a program. So, to express such properties in terms of equivalent JML annotations, we need a path semantics of the JML. This semantics is presented in this section.

3.1 Class in Isolation Semantics

In this section, we define, in term of *execution paths* the semantics of a class C *in isolation*. A class C is the description of a set of objects I_C, called the instances of C.

Definition 1 (Java Class) *A* class C *is a tuple* (V_C, M_C) *where* V_C *is the set of attributes of the class,* M_C *is the set of methods of the class. Within the set* M_C *of methods, we consider the following particular subsets:*

- $Cons_C \subseteq M_C$ *the set of constructors of the class.*
- $Helper_C \subseteq M_C$ *the set of helper methods of the class.*
- $\mathcal{PM}_C$ *the set of progress (side-effect) methods. The set of non-progress (pure) methods is denoted by* $\overline{\mathcal{PM}_C}$. *Notice that* $M_C = \mathcal{PM}_C \cup \overline{\mathcal{PM}_C}$.

Notice that, as said before, we do not consider in this paper the problem of inheritance. Intuitively, an *execution path* of a Java statement is the sequence of memory states reached during the execution of the statement. The structure of the memory model is not in the scope of the paper, but has been formally specified in [26] or [20]. Intuitively, it is composed of a *heap*, mapping variables to memory addresses and a *store*, mapping addresses to their values. JML predicates are pre/post predicates, therefore they are evaluated on two memory states. Let $P \in \mathcal{P}red$ be a JML predicate and let s_{pre} and s_{cur} be two memory states, we denote by $(s_{pre}, s_{cur}) \models P$ the satisfaction of P at these memory states. The values of the variables with an `\old` are in s_{pre} and the others are in s_{cur}. If P does not contain variables relating to a preceding state, i.e., no `\old`, then we simply denote $s_{cur} \models P$. Given a state s and a variable `a`, s(`a`) returns the value of `a` in the state s. The Java memory contains also an *execution stack* [18]. As in Logozzo [19], we do not explicitly use the *execution stack*, *heap* and *store* but we assume to have *special variables* to observe it.

Definition 2 (Special Variables)
Let s *be a memory state, assume to have the following special variables:*

- s(*excp*)*, a flag denoting that an exception has been thrown.*
- s(*stackHeight*)*, the height of the execution stack.*

```
public class Buffer {
  int len;
  byte [] status;
  byte [] buffer;
  int position = 0;
  boolean perso = false;
//@ ghost boolean trDepth = false;

//@ invariant position >= 0;

/*@ constraint position > \old(position)
  @ for write;
  @*/

/*@ normal_behavior
  @   requires perso == false;
  @   requires l > 0;
  @   assignable len,perso;
  @*/
void storeData(int l){
  len = l;
  perso = true;
}

 /*@pure@*/ byte [] getStatus(){
  return status;
}

 /*@pure@*/ int getBufferLess(){
  return len - buffer.length;
}

/*@ normal_behavior
  @   requires trDepth == false;
  @   requires perso == true;
  @   assignable buffer;
  @ also
  @ exceptional_behavior
  @   requires perso == false;
  @   assignable \nothing;
  @   signals (Exception e) true;
  @*/
void begin() throws Exception{
  if (perso == false) {
    throw new Exception();
  }
  buffer = new byte[len];
  //@ set trDepth = true;
}

/*@ normal_behavior
  @ requires trDepth == true;
  @ requires perso == true;
  @ assignable status,position;
  @*/
void commit(){
  status = buffer;
  position = 0;
  //@ set trDepth = false;
}

/*@ normal_behavior
  @ requires trDepth == true;
  @ requires perso == true;
  @ assignable position;
  @*/
void abort(){
  position = 0;
  //@ set trDepth = false;
}

/*@ normal_behavior
  @ requires trDepth == true;
  @ requires perso == true;
  @ requires position < len;
  @ assignable position;
  @ assignable buffer[position-1];
  @ ensures position <= len;
  @ ensures position == \old(position)+1;
  @*/
void write(byte b){
  buffer[position] = b;
  position++;
}

}
```

Figure 1: Class `Buffer`

- $s(\mathsf{curMethod})$, *the current method (the method on the top of the stack).*
- $s(\mathsf{curInstance})$, *the pointer on the current object.*

Intuitively, we define a path of a Java statement T as a sequence of states that are reached during the execution of T. For that, we assume a transition relation $\rightarrow$ associated to each Java statement T. Readers can find an example of such a relation in [12] for a sequential Java core.

Definition 3 (Java Statement Path) *Let s_0 be a Java state and let T be a Java statement, the* path *of T, denoted $s_0[T]$ is either:*

- *if T terminates, the* ***finite sequence*** *σ of states $s_0, s_1, s_2, \ldots, s_n$ such that $\forall i.(0 \leq i < n \Rightarrow (s_i \rightarrow s_{i+1}))$; or*
- *if T diverges, the* ***infinite sequence*** *σ of states $s_0, s_1, s_2, \ldots$ such that $\forall i \geq 0.(s_i \rightarrow s_{i+1})$.*

In the rest of the paper, we denote by s_i the i^{th} state of the execution path σ. We denote by ϵ the empty path and given an execution path σ the *path suffix* σ_i denotes the infinite path $s_i, s_{i+1}, s_{i+2} \ldots$ and the *path segment* σ_i^j denotes the finite path $s_i, s_{i+1}, s_{i+2} \ldots s_j$. A predicate P without `\old` holds on σ_i and σ_i^j if $s_i \models P$ as in LTL logic [23]. Given a finite path $\sigma = s_0, \ldots, s_n$ and another path $\sigma' = s_a, s_b, \ldots$, the sequential composition of σ and σ', denoted σ, σ' is equal to the path $s_0, \ldots, s_n, s_a, s_b, \ldots$.

As explained in Section 1, we aim at verifying that a property of a class C is satisfied by any instances of C and by any programs using the class C.

Therefore, we introduce a class in isolation semantics, handling all potential executions of all instances of a class C.

As explained in Sect. 2, JML considers only *visible* states, i.e., states before the invocation of a non-helper method or after the termination of a non-helper method. Therefore, we define the notions of pre- and post-states for a method m as follows.

Definition 4 ***(Pre-, Post-, Matching Post- and Inner-state of a Method)*** *Let σ be a path, let s_i be its i^{th} state (with $i > 0$) and let m be a method. We say that s_i is a* pre-state *of m, denoted $s_i \models \mathsf{pre}(m)$ if:*

$$s_i(\mathsf{curMethod}) = m \quad and$$
$$s_i(\mathsf{stackHeight}) = s_{i-1}(\mathsf{stackHeight}) + 1$$

s_i is a post-state *of m, denoted $s_i \models \mathsf{post}(m)$ if:*

$$s_i(\mathsf{curMethod}) = m \quad and$$
$$s_i(\mathsf{stackHeight}) - 1 = s_{i+1}(\mathsf{stackHeight})$$

Given a pre-state s_i of m, a state s_j, where $j > i$ is its matching post-state *s_j, denoted $s_j \models \mathsf{post}_{s_i}(m)$, if:*

$$s_j \models \mathsf{post}(m) \ \wedge \ s_j(\mathsf{stackHeight}) = s_i(\mathsf{stackHeight})$$
$$\wedge \forall k.(i < k < j \Rightarrow (s_k(\mathsf{stackHeight}) \geq s_i(\mathsf{stackHeight}))).$$

Let i be the index of the pre-state of m, let j be the index of its matching post-state, a state s_k is an visible inner-*state of m, denoted $s_k \models \mathsf{inner}_{s_i}(m)$ if $i \geq k \geq j$. If m does not terminate, s_k is an* inner-*state if $i \geq k$.*

Then, we define the notion of *visible states* following [16].

Definition 5 (Visible States)
Given a Java execution path σ, the state s_i is a visible state *for an instance i_C of the class C, denoted $s_i \models \mathsf{visible}(i_C)$ either if*

- *s_i is the post-state of a* constructor *of i_C,*

$$s_i \models \mathsf{post}(m) \ \wedge \ m \in Cons_C \ \wedge$$
$$s_i(\mathsf{curInstance}) = i_C, \text{ or}$$

- s_i *is the pre-state of a* non-helper *method invoked on* i_C*,*

$$s_i \models \mathsf{pre}(m) \ \wedge \ m \in M_C \ \wedge \ m \notin Cons_C \ \wedge \\ m \notin Helper_C \ \wedge \ s_i(\mathsf{curInstance}) = \mathsf{i}_C, \text{ or}$$

- s_i *is the post-state of a* non-helper *method invoked on* i_C*.*

$$s_i \models \mathsf{post}(m) \ \wedge \ m \in M_C \ \wedge \ m \notin Cons_C \ \wedge \\ m \notin Helper_C \ \wedge \ s_i(\mathsf{curInstance}) = \mathsf{i}_C.$$

Therefore, for easily reasoning about JML, we define the notion of *visible state execution path* as follows:

Definition 6 (Visible State Abstraction and Visible State Execution Path) *Let* σ *be an execution, we define the* visible state abstraction *for an instance* i_C*, denoted* $\mathsf{vsa}_{\mathsf{i}_C}(\sigma)$*, by:*

- $\mathsf{vsa}_{\mathsf{i}_C}(\epsilon) = \epsilon$
- *if* $s_0 \models \mathsf{visible}(\mathsf{i}_C)$ *then* $\mathsf{vsa}_{\mathsf{i}_C}(\sigma) = s_0, \mathsf{vsa}_{\mathsf{i}_C}(\sigma_1)$ *else* $\mathsf{vsa}_{\mathsf{i}_C}(\sigma) = \mathsf{vsa}_{\mathsf{i}_C}(\sigma_1)$.

Given a Java statement S*, we define the visible state execution path of* S *on a state* s_0*, denoted* $s_0[S]_{\mathsf{i}_C}$*, as follows:*

$$s_0[S]_{\mathsf{i}_C} =_{def} \mathsf{vsa}_{\mathsf{i}_C}(s_0[S])$$

Notice that the visible state abstraction hides:

- The details of the C method's body execution.
- The invocations of helper methods.
- The invocations of methods of other classes (both methods of other classes invoked by C and by the environment of C).

Let Σ be a set of paths, the visible state abstraction of Σ w.r.t. an instance i_C, denoted $\mathsf{vsa}_{\mathsf{i}_C}(\Sigma)$ is defined as the set of all the abstractions of the paths of Σ, i.e.,

$$\mathsf{vsa}_{\mathsf{i}_C}(\Sigma) = \{\mathsf{vsa}_{\mathsf{i}_C}(\sigma) | \sigma \in \Sigma\}.$$

Then, we define the semantics of an instance i_C in isolation, denoted Σ_{i_C}. The semantics of i_C captures all the potential executions of i_C. So, it is intuitively the set of all paths Σ_{i_C} starting at invocation of a constructor creating i_C, followed by an arbitrary number of invocations on i_C of the methods of C within their preconditions.

Definition 7 (Instance Semantics) *Let* i_C *be an instance of* $C = (V_C, M_C)$*, we denote* Σ_{i_C} *the set of executions iteratively defined as follows:*

- $\epsilon \in \Sigma_{\mathsf{i}_C}$.
- *Let* s_0 *be a state, let* $m \in Cons_C$*. If* $s_0 \models \texttt{requires}(m)$ *then*

$$s_0[(m, \mathsf{i}_C)]_{\mathsf{i}_C} \in \Sigma_{\mathsf{i}_C}.$$

- *Let* $\sigma \in \Sigma_{\mathsf{i}_C}$ *be a finite execution and* s_n *be its last state. Let* s_{n+1} *be a state such that* $\forall v \in V_C.(s_n(v) = s_{n+1}(v))$*. Let* m *such that* $m \in M_C \wedge m \notin Cons_C \wedge m \notin Helper_C$*. If* $s_{n+1} \models \texttt{requires}(m)$ *then:*

$$(\sigma, (s_{n+1}[(m, \mathsf{i}_C)]_{\mathsf{i}_C})) \in \Sigma_{\mathsf{i}_C}.$$

The class semantics of a class C is defined as the set of all executions of its instances.

Definition 8 (Class semantics) *Given a class* C*, let* I_C *be the set of instances of* C*. We define* Σ_C*, the semantics of the class* C*, by:*

$$\Sigma_C =_{def} \bigcup_{i_C \in I_C} \Sigma_{i_C}$$

3.2 JML Semantics

To express temporal properties by JML annotations, we need an execution semantics of JML annotations. To our knowledge, JML semantics has been given in terms of wp-calculus (see for example [20]), but never in terms of properties of the execution paths. We propose in this section a semantics for the `invariant` clauses, `constraint` clauses and a `behavior` specification.

Definition 9 (Path Execution Semantics of JML annotations) *Given a set of executions* Σ_C *of a class in isolation, the* path execution semantics of JML annotations *is displayed in Fig. 2.*

The semantics is given w.r.t. the definition in [16]. It must be understood as follows[2].

- `Invariant`: The invariant must be satisfied by each visible state.
- `Constraint`: For the body of each method included in the `for` clause, the constraint must hold between the pre-state and the post-state, but also between all visible states that arise during the execution of the method, i.e., all *inner* states of the method.
- `Behavior` method specification: each specification of a method can be desugared as a `behavior` specification [16]. This JML specification is interpreted on a path as follows. i If the predicate P of the `requires` clause is satisfied by the pre-state of the method m, that implies:
  - If the predicate D of the `diverges` is satisfied on the pre-state, then if the method terminates, i.e., the method has a post-state, the predicate Q of the `ensures` clause must be satisfied between the pre-state and the post-state if it is a normal termination ($s_j(\mathsf{excp}) = false$). Otherwise, i.e., if it is an exceptional termination, the predicate R must be satisfied.
  - If the predicate D of the `diverges` is not satisfied on the pre-state, then the method *must* terminate, i.e., the method *must* have a post-state. Moreover, if it is a normal termination the predicate Q must be satisfied, and the predicate R must be satisfied otherwise.

  Notice that in each case, only attributes within the list A of the `assignable` clause can be modified ($\forall a.(a \in V_S \wedge a \notin A \Rightarrow (s_i(a) = s_j(a)))$).

[2]The definitions of `constraint` and `assignable` are proposed accordingly to the semi-formal description in [16]. Notice that, for technical reasons, an alternative semantics of these clauses has been implemented in some tools

`//@ invariant` I`;` $\equiv_{def}$ $\forall \sigma \in \Sigma_C \,.\, \forall i \geq 0 \,.\, \sigma_i \models I$

`//@ constraint` H `for` M`;` $\equiv_{def}$

$$\begin{array}{l}\forall \sigma \in \Sigma_C \,.\, \forall i \geq 0 \,.\, \forall m \in M. \\ \quad (s_i \models \mathsf{pre}(m) \Rightarrow \\ \quad\quad (\forall k_1, k2.(i \leq k_1 < k_2 \\ \quad\quad\quad \wedge s_{k_1} \models \mathsf{inner}_{s_i}(m) \wedge s_{k_2} \models \mathsf{inner}_{s_i}(m) - \Rightarrow \\ \quad\quad\quad\quad (s_{k_1}, s_{k_2}) \models H)))\end{array}$$

```
/*@ behavior;
  @ requires P;
  @ diverges D;
  @ assignable A;
  @ ensures Q;
  @ signals
  @ (Exception e) R;
  @*/
m()
```

$\equiv_{def}$

$$\begin{array}{l}\forall \sigma \in \Sigma_C \,.\, \forall i \geq 0 \,.(\\ \quad ((\sigma_i \models (P \wedge \neg D) \wedge \sigma_i \models \mathsf{pre}(m)) \Rightarrow \\ \quad\quad \exists j > i.(\\ \quad\quad\quad (s_j \models \mathsf{post}_{s_i}(m) \wedge s_j(\mathsf{excp}) = false \\ \quad\quad\quad\quad \wedge (s_i, s_j) \models Q \\ \quad\quad\quad\quad \wedge \forall a.(a \in V_S \wedge a \notin A \Rightarrow (s_i(a) = s_j(a)))) \\ \quad\quad\quad \vee \\ \quad\quad\quad (s_j \models \mathsf{post}_{s_i}(m) \wedge s_j(\mathsf{excp}) = true \\ \quad\quad\quad\quad \wedge (s_i, s_j) \models R \\ \quad\quad\quad\quad \wedge \forall a.(a \in V_S \wedge a \notin A \Rightarrow (s_i(a) = s_j(a))))) \\ \quad \wedge \\ \quad ((\sigma_i \models (P \wedge D) \wedge \sigma_i \models \mathsf{pre}(m)) \Rightarrow \\ \quad\quad \forall j > i.(\\ \quad\quad\quad (s_j \models \mathsf{post}_{s_i}(m) \wedge s_j(\mathsf{excp}) = false \Rightarrow \\ \quad\quad\quad\quad (s_i, s_j) \models Q \\ \quad\quad\quad\quad \wedge \forall a.(a \in V_S \wedge a \notin A \Rightarrow (s_i(a) = s_j(a)))) \\ \quad\quad\quad \wedge \\ \quad\quad\quad (s_j \models \mathsf{post}_{s_i}(m) \wedge s_j(\mathsf{excp}) = true \Rightarrow \\ \quad\quad\quad\quad (s_i, s_j) \models R. \\ \quad\quad\quad\quad \wedge \forall a.(a \in V_S \wedge a \notin A \Rightarrow (s_i(a) = s_j(a))))))))\end{array}$$

Figure 2: Path execution semantics of JML annotations

4. LIVENESS PROPERTIES VERIFICATION

This section deals with the verification of liveness properties on the execution semantics Σ_C of a class C. For that, we presents in Section 4.1 a *liveness primitive operator* Loop. Under a *progress hypothesis* on the environment presented in Section 4.2, the satisfaction of the Loop operator can be ensured by appropriates JML annotations. This result is established in a theorem given in Section 4.3.

4.1 The Loop Primitive

In this section, Q denotes a JML predicate, M denotes a subset of $\mathcal{PM}_C$, and V denotes a JML expression returning an integer. The $\mathsf{Loop}(Q, V, M)$ primitive is satisfied by an execution if, after any states of the execution satisfying Q, a state where $\neg Q$ holds must *eventually* be reached. Besides the predicate Q marking the loop entry condition we also require, to prove the termination of the loop, a variant V and a set of methods $M \subseteq \mathcal{PM}_C$.

Definition 10 (Loop Primitive) $\mathsf{Loop}(Q, V, M)$[3] *holds on an execution* σ*, written* $\sigma \models \mathsf{Loop}(Q, V, M)$*, if*

$$\forall i.((i \geq 0 \wedge \sigma_i \models Q) \;\Rightarrow\; (\exists j.j > i \wedge \sigma_j \models \neg Q)).$$

If σ *is a finite execution of length* n*,* $\sigma \models \mathsf{Loop}(Q, V, M)$ *if*

$$\forall i.((0 \leq i \leq n \wedge \sigma_i \models Q) \;\Rightarrow\; (\exists j.i < j \leq n \wedge \sigma_j \models \neg Q)).$$

Notice that the variant V and the set M of methods do not appear in the above expression since they are only used to generate the appropriate proof obligations for the termination of the loop. For the verification of the Loop operator, finite executions are viewed as infinite executions by infinitely repeating the last state of the execution. The infinite extension of a finite execution is the following.

Definition 11 (Infinite Extension of Finite Execution) *Let* σ *be a finite execution* $s_0, s_1, s_2 \ldots, s_n$*. We extend it to the infinite sequence* σ' *such that* σ' *is* $s_0, s_1, s_2 \ldots, s_n, s_n, \ldots$.

The infinite extensions of finite executions are suitable for verifying the Loop primitive.

Lemma 1 *Let* σ *be a finite execution,* σ' *be the infinite extension of* σ*. We have*

$$\sigma' \models \mathsf{Loop}(Q, V, M) \quad \Leftrightarrow \quad \sigma \models \mathsf{Loop}(Q, V, M)$$

Notice that Σ_C contains all *potential* executions of instances of C. We address the verification of a particular subset of Σ_C that satisfies a *progress hypothesis*.

4.2 Progress Hypothesis PH

Using the semantics of LTL [23], Hypothesis $PH(Q, M)$ is expressed by the LTL operators G^∞ ("almost everywhere") and F^∞ (*"infinitely often"*).

Intuitively, $\mathsf{G}^\infty P$ means that after a finite number of states, the property P holds forever. The semantics of G^∞ is the following

$$\sigma_i \models G^\infty P \equiv_{def} \exists j \geq i.(\forall k.(k \geq j \Rightarrow \sigma_k \models P)).$$

Given a predicate P, the formula $F^\infty P$ means that at any state of the execution, there always exists a future state verifying P. Formally,

$$\sigma_i \models F^\infty P \equiv_{def} \forall j \geq i.(\exists k.(k \geq j \wedge \sigma_k \models P)).$$

In order to verify $\Sigma_C \models \mathsf{Loop}(Q, V, M)$, we need to assume progress of the environment, i.e., the environment invokes

[3]Notice that Loop(Q,V,M) semantics corresponds to LTL property $\mathsf{GF}\neg Q$.

the methods of the subset M of the *progress* methods. Two behaviors of the environment are allowed:

- The environment calls methods in M infinitely often.
- The environment performs a finite number of invocations of methods in M until a state i such that any state of σ_i satisfies $\neg Q$.

Therefore, we define the progress hypothesis $PH(Q, M)$ as follows.

Definition 12 (Progress Hypothesis $PH(Q, M)$)

$$(\mathsf{G}^{\infty}\neg Q) \vee (\mathsf{F}^{\infty}\mathsf{pre}(M)) \qquad \text{(PH(Q,M))}$$

where $\mathsf{pre}(M)$ *denotes the predicate* $\bigvee_{m\in M} \mathsf{pre}(m)$.

We denote $\Sigma_{C/PH(Q,M)}$ the subset of executions of Σ_C satisfying $PH(Q, M)$.

Definition 13 (Class under $PH(Q, M)$) $\Sigma_{C/PH(Q,M)}$ *is the set of execution defined as follows.*

$$\Sigma_{C/PH(Q,M)} = \{\sigma | \sigma \in \Sigma_C \wedge \sigma \models PH(Q, M)\}.$$

In the next section we show how to use appropriate JML annotations for establishing that $\Sigma_{C/PH(Q,M)} \models \mathsf{Loop}(Q, V, M)$.

4.3 Annotations for the Loop operator

Verification of the Loop primitive is quite similar to a termination proof, since we have to show that as long as Q it must always be possible to invoke a method of M and methods in M must decrease a well founded variant V. Here we propose proof obligations – inspired from [9] – expressed as JML annotations. These proof obligations guarantee the satisfaction of the Loop primitive by an execution satisfying the hypothesis $PH(Q, M)$.

Let $\mathsf{Loop}(Q, V, M)$ be the Loop primitive. Let $\mathcal{A}_{1-5}$ be the following set of JML annotations.

`//@ invariant` V `>= 0;` $(\mathcal{A}_1)$

`//@ constraint` Q `==>` V `< \old(`V`) for` M`;` $(\mathcal{A}_2)$

`//@ constraint` Q `==>` V `<= \old(`V`) ;` $(\mathcal{A}_3)$

`//@ invariant` Q `==>` $\bigvee_{m\in M}$ $\mathsf{requires}(m)$ $(\mathcal{A}_4)$

`//@ invariant` Q `==>` $\bigwedge_{m\in M_C}$ $(\mathsf{requires}(m)$ `==>` $\mathsf{!diverges}(m))$`;` $(\mathcal{A}_5)$

Intuitively, $\mathcal{A}_{1-5}$ could be understood as follows.

$\mathcal{A}_1$ The variant V actually is greater than zero, i.e., it is an expression over a well-founded set.

$\mathcal{A}_2$ As long as Q holds, when a method in M is executed, the variant V must decrease. It ensures the progress when the environment satisfies $PH(Q, M)$ (livelock-freeness).

$\mathcal{A}_3$ As long as Q holds, when a method of C is executed, the variant V must not increase.

$\mathcal{A}_4$ As long as Q holds there always should be a method in M that might be called, i.e., its precondition holds. This ensures the deadlock-freeness of the system.

$\mathcal{A}_5$ As long as Q holds, all callable methods must not diverge. This ensures the non-divergence of the system.

Hypothesis $PH(Q, M)$ is the disjunction of $(\mathsf{F}^{\infty}\mathsf{pre}(M)$ and $\mathsf{G}^{\infty}(\neg Q)$, therefore, for each of these hypothesis, we show respectively in Lemma 2 and Lemma 3 that, assuming that the code of the class is correct w.r.t. the annotations $\mathcal{A}_{1-5}$ $(C : \mathcal{A}_{1-5})$, the satisfaction of Loop (Q, V, M) is established on Σ_C.

Lemma 2 *If* $C : \mathcal{A}_{1-5}$ and $\sigma \in \Sigma_C$ and $\sigma \models (\mathsf{F}^{\infty}\mathsf{pre}(M))$ *then* $\sigma \models \mathsf{Loop}(Q, V, M)$.

Lemma 3 *If* $C : \mathcal{A}_{1-5}$ and $\sigma \in \Sigma_C$ and $\sigma \models \mathsf{G}^{\infty}(\neg Q)$ *then* $\sigma \models \mathsf{Loop}(Q, V, M)$.

A consequence of Lemma 2 and Lemma 3 is the following theorem.

Theorem 1

If $C : \mathcal{A}_{1-5}$ then $\Sigma_{C/PH(Q,M)} \models \mathsf{Loop}(Q, V, M)$.

An interesting property is obtained when $M = \mathcal{PM}_C$. In this particular case, Hypothesis $PH(Q, M)$ is not only sufficient, but also necessary.

Proposition 1 *When* $M = \mathcal{PM}_C$, *given* $\sigma \in \Sigma_C$, $\sigma \models \mathsf{Loop}(Q, V, M)$ *and* $C : \mathcal{A}_{1-5}$ *imply that* $\sigma \models PH(Q, M)$.

We now show how liveness properties (expressed here in JTPL) can be embedded into a Loop primitive.

5. JML ANNOTATION GENERATOR TOOL

The generation of annotations for safety properties in [25] and of liveness properties presented in Sect. 4 is implemented in a tool, called JAG (for JML Annotation Generator) [10].

The JAG tool takes as an input a formula expressed in JML Temporal Pattern Logic (JTPL), first introduced in [25]. A JTPL formula is a combination of JML predicates, *events* and temporal operators. Using JTPL formulae, one can express, on the example of the `Buffer` (see Fig. 1 Sect. 2), the following properties:

1. After the invocation of `storeData` (**after** `storeData` **called**), the variable `perso` is **always** `true`, expressed in JTPL as follows.

   after `storeData` **called always** `perso;` (S)

2. After starting a transaction, i.e., the normal termination of the method `begin` (**after** `begin` **normal**), a state where `trDepth` is `false` must eventually be reached.

   after `begin` **normal eventually** `!trDepth`
   under variant `getBufferLess()`
   for `begin, commit, abort, write .` (L)

Notice that in Property L, the event is `begin` **normal** and not `begin` **called** since a buffer transaction starts only when the method `begin` terminates normally. Notice also that since Property L is a liveness property, the user gives a variant and a set of progress methods with the JTPL clause `under_variant ... for`.

The result of the translation of Properties S and L is displayed in Fig. 3.

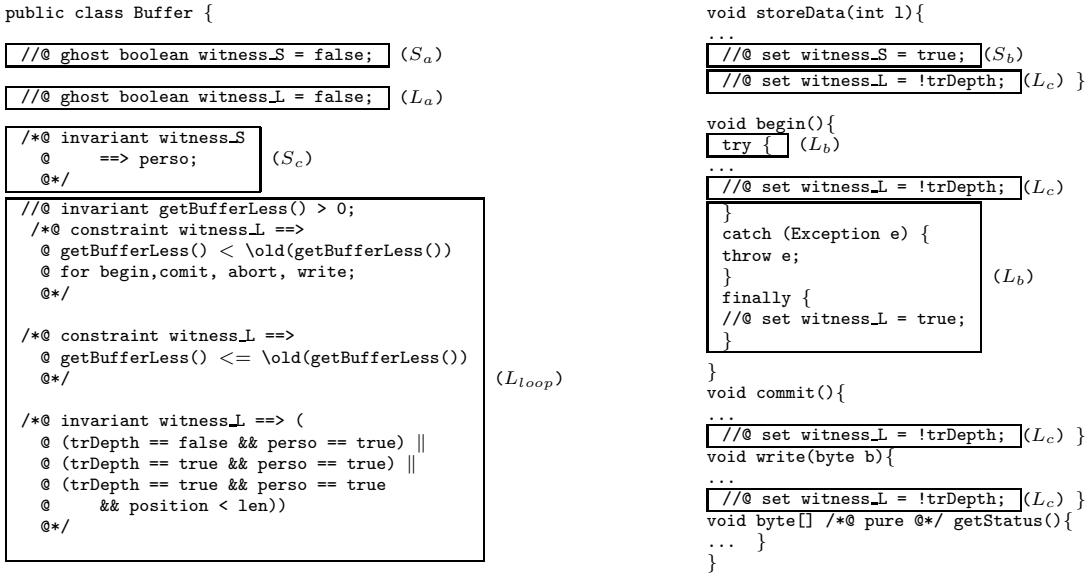

Figure 3: Buffer with generated annotations

1. First, JAG generates a *ghost* boolean variable for observing the occurrences of the events of the temporal properties. These ghost variables are assigned w.r.t. the events occurring in the formula.

   Example 1 (Ghost Variables Generation for S) *The* `ghost` *variable* `witness_S`*, corresponds to the event* `storeData` ***called*** *of S. It is initially declared with the value* `false` *(see Annotation S_a in Fig. 3) and it is set to* `true` *when the method* `storeData` *is called (see annotation S_b). So, in each state after the event* `storeData` ***called****, the value of the ghost variable* `witness_S` *is* `true`*, i.e.,* `witness_S` *is true exactly with the scope of the property.*

   Example 2 (Ghost Variables Generation for L) *The* `ghost` *variable* `witness_L`*, corresponding to the event* `begin` ***normal*** *of the temporal property L is also declared with the value false (Annotation L_a in Fig. 3). The ghost variable* `witness_L` *is assigned using a* `try...catch...finally` *statement (see annotation L_b). Notice that, in case of exception, the caught exception is re-thrown, the execution does not go into the finally block, the reader can see that* `witness_L` *is set to* `true` *only when* `begin` ***normal*** *occurs. The* `ghost` *variable* `witness_L` *is set to* `false` *again by adding to each method a set statement (annotation L_c).*

2. Second, it generates an invariant to ensure the satisfactions of a safety property.

   Example 3 (Invariant Generation for S) *The invariant for S is displayed in Fig. 3 (annotation S_c). It means that when the variable* `witness_S` *is true, i.e., after the first occurrence of* `storeData` ***called****, the predicate* `(perso == true)` *must be* `true` *- the definition of Property S.*

3. Finally JAG translates each liveness property into a `Loop` primitive and generates the corresponding JML annotations.

   Example 4 (Generation of annotations for L) *The JML primitive corresponding to L is*

   `Loop(witness_L, getBufferLess(), {begin, commit, abort, write})`

   The corresponding annotations are displayed in Fig. 3 (see Annotations L_{loop}).

   Notice that, since no method of `Buffer` diverges, Annotation $\mathcal{A}_5$ does not appear.

The tool is able to keep the trace of the generated annotations, i.e., it is possible, given a generated annotation, to find the original intermediate primitive and the original temporal property. Since the generated output file contains standard JML annotations, it can be used with other JML tools [7] to validate or prove the temporal formulae. In particular, we have successfully used it for the following purposes.

- **Verification of the correctness of the Java code w.r.t. the JML annotations** with the proof obligation generators Jack [8] and Krakatoa [20].
- **Validation of a JML model** with JML-TT [5];
- **Formal verification of a JML model** with the JML2B method [2];
- **Test generation and Runtime Assertion Checking** with the test generators Tobias [17], Jartege [22] and JML-TT [4].

Test generation and Runtime Assertion Checking using JAG has been studied on a industrial Javacard application [3].

6. CONCLUSION AND FUTURE WORKS

This paper presents a way to verify liveness properties on Java classes in isolation by generating appropriate JML annotations. This requires that the user specifies a variant for the verification of a Loop primitive to which liveness properties are reduced. The generated JML annotations are verified (or validated) with any tools handling JML. The JAG tool implements this translation. It has been used for several toy examples and a Java Card Electronic Purse Specification (over 500 lines of JML).

To the best of our knowledge, this is the first attempt to verify liveness properties for potentially infinite-state systems using a translation into JML. For finite state systems, liveness properties expressed in LTL are usually verified automatically by model checkers such as SPIN [11]. For infinite state systems, model checking is used on liveness preserving abstractions.

Currently we are working on extensions of JAG to other temporal properties. In particular, we currently address the verification of properties expressed by Büchi automata. Then, the Büchi acceptance condition is checked using Loop primitives introduced in this paper. The second challenge is, assuming that a liveness is established on the class in isolation, to provide techniques for verifying that the (single- or multi-threaded) environment effectively satisfies $PH(Q, M)$.

Acknowledgment: We like to thank Marieke Huisman for her interesting and helpful comments and suggestions to improve this work. Further, we acknowledge all the anonymous referees for their corrections, comments and advices.

7. REFERENCES

[1] J. Andronick, B. Chetali, and O. Ly. Using Coq to verify Java Card Applet Isolation Properties. In *16th International Conference on Theorem Proving in Higher Order Logics (TPHOLs'2003)*, 2003.

[2] F. Bouquet, F. Dadeau, and J. Groslambert. Checking JML specifications with B machines. In *ZB'05*, volume 3455 of *LNCS*, pages 435–454. Springer-Verlag, 2005.

[3] F. Bouquet, F. Dadeau, J. Groslambert, and J. Julliand. Safety property driven test generation from JML specifications. In *FATES/RV'06*, LNCS, pages 225–239. Springer-Verlag, 2006. To appear.

[4] F. Bouquet, F. Dadeau, and B. Legeard. Automated Boundary Test Generation from JML Specifications. In *FM'06*, volume 4085 of *LNCS*, pages 428–443. Springer-Verlag, 2006.

[5] F. Bouquet, F. Dadeau, B. Legeard, and M. Utting. JML-Testing-Tools: a symbolic animator for JML specifications using CLP. In *TACAS'05 Tool session*, volume 3440 of *LNCS*, pages 551–556. Springer, 2005.

[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, Y. Cheon, D. Cok, M. Ernst, J. Kiniry, G.T. Leavens, K.R.M. Leino, and E. Poll. An Overview of JML Tools and Applications. In *FMICS 03*, volume 80 of *ENTCS*, pages 73–89. Elsevier, 2003.

[8] L. Burdy, A. Requet, and J.-L. Lanet. Java Applet Correctness: a Developer-Oriented Approach. In *FM'03*, number 2805 in LNCS, pages 422–439. Springer, 2003.

[9] R.M. Burstall. Program Proving as Hand Simulation with a Little Induction. *Information Processing*, pages 308–312, 1974.

[10] A. Giorgetti and J. Groslambert. JAG: JML Annotation Generation for Verifying Temporal Properties. In *FASE*, LNCS, pages 373–376. Springer, 2006.

[11] G.J. Holzmann. The Model Checker SPIN. In *IEEE Trans. on Software Engineering*, volume 23-5, pages 279–295, 1997.

[12] A. Igarashi, B. Pierce, and P. Wadler. Featherweight Java: A minimal core calculus for Java and GJ. In *OOPSLA*, volume 34(10), pages 132–146. ACM, 1999.

[13] B. Jacobs, C. Marché, and N. Rauch. Formal Verification of a Commercial Smart Card Applet with Multiple Tools. In *AMAST'04*, number 3116 in LNCS, pages 21–22. Springer, 2004.

[14] O. Kouchnarenko, J. Groslambert, and J. Julliand. JML-based Verification of Liveness Properties on a Class. Technical Report RR2006-7, LIFC, 2006.

[15] L. Lamport. Proving the Correctness of Multiprocess Programs. In *IEEE Transactions on Software Engineering*, volume 3(2), pages 125–143, 1977.

[16] G.T. Leavens, E. Poll, C. Clifton, Y. Cheon, C. Ruby, D.R. Cok, and J. Kiniry. JML Reference Manual. Department of Comp. Science, Iowa State University. Available from http://www.jmlspecs.org, 2003.

[17] Y. Ledru, L. du Bousquet, O. Maury, and P. Bontron. Filtering TOBIAS Combinatorial Test Suites. In *FASE 2004*, volume 2984 of *LNCS*, pages 281–294. Springer-Verlag, 2004.

[18] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. The Java Series. Addison-Wesley, Reading, MA, USA, 1997.

[19] F. Logozzo. Class Invariants as Abstract Interpretation of Trace Semantics. *Computer Languages, Systems and Structures*, 2005.

[20] C. Marché, C. Paulin-Mohring, and X. Urbain. The Krakatoa tool for certification of Java/Java Card programs annotated in JML. *Journal of Logic and Algebraic Programming*, 58(1-2):89–106, 2004.

[21] B. Meyer. *Object-Oriented Software Construction*. Prentice Hall, 2nd rev. edition, 1997.

[22] 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.

[23] A. Pnueli. The Temporal Logic of Program. In *18th Ann. IEEE Symp. on foundations of computer science*, pages 46–57, 1977.

[24] Robby, E. Rodríguez, M. Dwyer, and J. Hatcliff. Checking Strong Specifications Using an Extensible Software Model Checking Framework. In *TACAS 2004*, volume 2988, pages 404–420. Springer, 2004.

[25] K. Trentelman and M. Huisman. Extending JML Specifications with Temporal Logic. In *AMAST'02*, number 2422 in LNCS, pages 334–348. Springer, 2002.

[26] J. van den Berg, M. Huisman, B. Jacobs, and E. Poll. A Type-Theoretic Memory Model for Verification of Sequential Java Programs. In *WADT*, volume 1827 of *LNCS*, pages 1–21. Springer, 1999.