JML-based Verification of Liveness Properties on a Class in isolation
Julien Groslambert, Jacques Julliand, Olga Kouchnarenko

To cite this version:

HAL Id: hal-00561442
https://hal.archives-ouvertes.fr/hal-00561442
Submitted on 1 Feb 2011

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L’archive ouverte pluridisciplinaire HAL, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d’enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.
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 project¹ (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

\[ \text{"After the invocation of the method } m \text{ the property } P \text{ must be satisfied in all the states". } \]  
\[ \text{"After the non-exceptional termination of the method } m \text{ a state where } P \text{ holds must inevitably be reached in the future".} \]

Following [15], Property \( S \) is a safety property, expressing that \textit{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 \( m \), \textit{something good} – a state where \( P \) holds – must \textit{invariably} happen in the future.

A key concept for reasoning modularly about safety properties is the notion of \textit{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 \textit{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 \textit{Loop}, inspired from [9], for expressing liveness properties; (3) a way to verify on a class in isolation liveness properties expressed with the \textit{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 \textit{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-
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 written in Java Card [6, 13]. JML is developed following the semantics of Java annotations. Section 6 presents the main JML semantics of methods of the class. Within the set \(M_{C}\) of methods, we consider the following particular subsets:

- \(Con_{C}\subseteq M_{C}\) the set of constructors of the class.
- \(Helper_{C}\subseteq M_{C}\) the set of helper methods of the class.
- \(PM_{C}\) the set of progress (side-effect) methods. The set of non-progress (pure) methods is denoted by \(PM_{C}\).

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 \(P\in \mathcal{P}rede\) 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 \(\`\text{old}\) 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 \(\`\text{old}\), 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:

- \(s(\text{excp})\), a flag denoting that an exception has been thrown.
- \(s(\text{stackHeight})\), the height of the execution stack.
public class Buffer {
    int len;
    byte[] status;
    byte[] buffer;
    int position = 0;
    boolean perso = false;
    //@ invariant boolean trDepth = false;
    //@ invariant position >= 0;
    //@ constraint position > \old(position)
    //@ write;
    @*
    //@ normal_behavior
    @ requires perso == false;
    @ requires i > 0;
    @ assignable len,perso;
    @*/
    void storeData(int l){
        len = l;
        perso = true;
    }
    @*pure*/
    return status;
    @*/
    void commit(){
        status = buffer;
        position = 0;
    }
    @*pure*/
    int getIntBufferLess(){
        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 abort(){
        position = 0;
        //@ set trDepth = false;
    }
    @/
    @ normal_behavior
    @ requires trDepth == true;
    @ requires perso == true;
    @ assignable position;
    @*/
    void write(byte b){
        buffer[position] = b;
        position++;
    }
    @*pure*/
    @ assignable len;
    @ requires position < len;
    @ assignable buffer[position-1];
    @ ensures position <= len;
    @ ensures position == \old(position)+1;
    @*/
    @ requires trDepth == false;
    @ assignable trDepth;
    @*
    //@ set trDepth = false;
    @*/
    @ normal_behavior
    @ requires trDepth == true;
    @ requires perso == true;
    @ assignable position;
    @*/
    void begin() throws Exception{
        if (perso == false) {
            throw new Exception();
        } else {
            buffer = new byte[len];
            //@ set trDepth = true;
        }
    }
}

Figure 1: Class Buffer

- \textit{s(curMethod)}\textup{,} the current method (the method on the top of the stack).
- \textit{s(curInstance)}\textup{,} 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 \( \sigma \) of states \( s_0, s_1, s_2, \ldots, s_n \) such that \( \forall i \in [0, n) \Rightarrow (s_i \rightarrow s_{i+1}) \); or

- if \( T \) diverges, the infinite sequence \( \sigma \) 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 \( \sigma \). We denote by \( e \) the empty path and given an execution path \( \sigma \) the path suffix \( \sigma' \) denotes the infinite path \( s_i, s_{i+1}, s_{i+2} \ldots \) and the path segment \( \sigma_i \) denotes the finite path \( s_i, s_{i+1}, s_{i+2} \ldots s_j \). A predicate \( P \) without \( \old \) holds on \( \sigma \), and \( \sigma_i \) if \( s_i \models P \) as in LTL logic [23]. Given a finite path \( \sigma = s_0, \ldots, s_n \) and another path \( \sigma' = s_0, s_1, \ldots \), the sequential composition of \( \sigma \) and \( \sigma' \) denoted \( \sigma \circ \sigma' \) is equal to the path \( s_0, \ldots, s_n, s_1, \ldots \).

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 \( \sigma \) 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 \( s_i \models \text{pre}(m) \) if:

\[
s_i(\text{curMethod}) = m \quad \text{and} \quad s_i(\text{stackHeight}) = s_{i-1}(\text{stackHeight}) + 1
\]

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

\[
s_i(\text{curMethod}) = m \quad \text{and} \quad s_i(\text{stackHeight}) - 1 = s_{i+1}(\text{stackHeight})
\]

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

\[
s_j \models \text{post}(m) \land s_j(\text{stackHeight}) = s_i(\text{stackHeight}) \land \forall k. (i < k < j \Rightarrow (s_k(\text{stackHeight}) \geq s_i(\text{stackHeight})))
\]

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 \( s_k \models \text{inner}(m) \) 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 \( \sigma \), the state \( s_i \) is a visible state for an instance \( iC \) of the class \( C \), denoted \( s_i \models \text{visible}(iC) \) either if

- \( s_i \) is the post-state of a constructor of \( iC \),

\[
s_i \models \text{post}(m) \land m \in \text{Cons_C} \land s_i(\text{curInstance}) = iC, \text{ or}
\]
• \( s_i \) is the pre-state of a non-helper method invoked on \( i_C \),
  \( s_i \models \text{pre}(m) \land m \in MC \land m \notin Cons_C \land m \notin Helper_C \land s_i(\text{curInstance}) = i_C \), or

• \( s_i \) is the post-state of a non-helper method invoked on \( i_C \),
  \( s_i \models \text{post}(m) \land m \in MC \land m \notin Cons_C \land m \notin Helper_C \land s_i(\text{curInstance}) = i_C \).

Definition 6 (Visible State Abstraction and Visible State Execution Path) Let \( \sigma \) be an execution, we define the visible state abstraction for an instance \( i_C \), denoted vsa\(_iC\)(\( \sigma \)), by:

- \( \text{vsa}_iC(\epsilon) = \epsilon \)
- \( \text{if} \ s_0 \models \text{visible}(i_C) \ \text{then} \ \text{vsa}_iC(\sigma) = s_0, \ \text{vsa}_iC(\sigma_1) \ \text{else} \ \text{vsa}_iC(\sigma) = \text{vsa}_iC(\sigma_1). \)

Given a Java statement \( S \), we define the visible state execution path of \( S \) on a state \( s_0 \), denoted \( s_0[S]_iC \), as follows:

\[
\text{vs}a_iC(\{s_0[S]_iC\}) = (\text{vs}a_iC(\{s_0[S]\}))
\]

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 \( \Sigma \) be a set of paths, the visible state abstraction of \( \Sigma \) w.r.t. an instance \( i_C \), denoted vsa\(_iC\)(\( \Sigma \)) is defined as the set of all the abstractions of the paths of \( \Sigma \), i.e.,

\[
\text{vsa}_iC(\Sigma) = \{\text{vsa}_iC(\sigma) | \sigma \in \Sigma\}.
\]

Then, we define the semantics of an instance \( i_C \) in isolation, denoted \( \Sigma_iC \). The semantics of \( i_C \) captures all the potential executions of \( i_C \). So, it is intuitively the set of all paths \( \Sigma_iC \) 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 \( C = (V_C, MC) \), we denote \( \Sigma_iC \) the set of executions iteratively defined as follows:

- \( \epsilon \in \Sigma_iC \).
- Let \( s_0 \) be a state, let \( m \in Cons_C \). If \( s_0 \models \text{requires}(m) \) then
  \( s_0[(m, i_C)]_{iC} \in \Sigma_iC \).
- Let \( \sigma \in \Sigma_iC \) 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 \( m \in MC \land m \notin Cons_C \land m \notin Helper_C \). If \( s_n+1 \models \text{requires}(m) \) then:
  \( (\sigma, (s_{n+1}[(m, i_C)]_{iC})) \in \Sigma_iC \).

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

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

\[
\Sigma_C = \text{def} \bigcup_{iC \in I_C} \Sigma_iC
\]

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 \( \Sigma_C \) 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:

- **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 (\( s_j(\text{exp}) = \text{false} \)). 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 (\( \forall a. (a \in V_S \land a \notin A \Rightarrow (s_i(a) = s_j(a))) \)).

2The 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; \[\forall \sigma \in \Sigma_C \cdot \forall i \geq 0 \cdot \sigma_i \models I\]

// constraint H for M; \[\forall \sigma \in \Sigma_C \cdot \forall i \geq 0 \cdot \forall m \in M.\]

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

/**
 * \(\forall \sigma \in \Sigma_C \cdot \forall i \geq 0 . (\sigma_i \models (P \land \neg D) \land \sigma_i \models \text{pre}(m)) \Rightarrow \exists j > i. (\sigma_j \models \text{post}_m(m) \land \sigma_j \models \text{excp}) = \neg false\)
 * \(\land (\sigma_i, \sigma_j) \models Q\)
 * \(\land \forall a. (a \in V_S \land a \notin A \Rightarrow (\sigma_i(a) = \sigma_j(a)))\)
 * \(\lor (\sigma_i \models (P \land D) \land \sigma_i \models \text{pre}(m)) \Rightarrow \forall j > i. (\sigma_j \models \text{post}_m(m) \land \sigma_j \models \text{excp}) = \neg false\)
 * \(\land (\sigma_i, \sigma_j) \models Q\)
 * \(\land \forall a. (a \in V_S \land a \notin A \Rightarrow (\sigma_i(a) = \sigma_j(a)))\)
 */

m()

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 \(\Sigma_C\) of a class C. For that, we present 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 appropriate 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 \(PM_C\), and \(V\) denotes a JML expression returning an integer. The 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 prove the termination of the loop, a variant \(V\) and a set of methods \(M \subseteq PM_C\).

Definition 10 (Loop Primitive) Loop\((Q, V, M)\)\(^3\) holds on an execution \(\sigma\), written \(\sigma \models \text{Loop}(Q, V, M)\), if

\[\forall i. ((i \geq 0 \land \sigma_i \models Q) \Rightarrow (\exists j > i \land \sigma_j \models \neg Q)).\]

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

\[\forall i. ((0 \leq i \leq n \land \sigma_i \models Q) \Rightarrow (\exists j < i \land \sigma_j \models \neg Q)).\]

Notice that the variant \(V\) and the set of methods \(M\) 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 \(\sigma\) be a finite execution \(s_0, s_1, s_2, \ldots, s_n\). We extend it to the infinite sequence \(\sigma'\) such that \(\sigma'\) is \(s_0, s_1, s_2, \ldots, s_n, \ldots\).

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

Lemma 1 Let \(\sigma\) be a finite execution, \(\sigma'\) be the infinite extension of \(\sigma\). We have

\[\sigma' \models \text{Loop}(Q, V, M) \iff \sigma \models \text{Loop}(Q, V, M)\]

Notice that \(\Sigma_C\) contains all potential executions of instances of \(C\). We address the verification of a particular subset of \(\Sigma_C\) 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 \(G^\infty\) (“almost everywhere”) and \(F^\infty\) (“infinitely often”).

Intuitively, \(G^\infty P\) means that after a finite number of states, the property \(P\) holds forever. The semantics of \(G^\infty\) 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,

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

In order to verify \(\Sigma_C \models \text{Loop}(Q, V, M)\), we need to assume progress of the environment, i.e., the environment invokes
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 \( \sigma \) such that any state of \( \sigma \) satisfies \( \neg Q \).

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

**Definition 12** (Progress Hypothesis \( PH(Q,M) \))

\[
(G^\infty \neg Q) \lor (F^\infty pre(M)) \quad (PH(Q,M))
\]

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

We denote \( \Sigma_{PH(Q,M)} \) the subset of executions of \( \Sigma \) satisfying \( PH(Q,M) \).

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

\[
\Sigma_{PH(Q,M)} = \{ \sigma | \sigma \in \Sigma \land \sigma \models PH(Q,M) \}.
\]

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

### 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 \( Loop(Q,V,M) \) be the Loop primitive. Let \( A_{1.5} \) be the following set of JML annotations.

\[
// \text{invariant } V >= 0; \quad (A_1)
\]

// \text{constraint } Q => V < \old(V) \text{ for } M; \quad (A_2)

// \text{constraint } Q => V <= \old(V) \text{ ;} \quad (A_3)

// \text{invariant } Q \Rightarrow \bigvee_{m \in M} \text{ requires(m)} \quad (A_4)

// \text{invariant } Q \Rightarrow \bigwedge_{m \in M} \text{ requires(m) } \Rightarrow ! \text{diverges(m)}; \quad (A_5)

Intuitively, \( A_{1.5} \) could be understood as follows.

\( A_1 \) The variant \( V \) actually is greater than zero, i.e., it is an expression over a well-founded set.

\( A_2 \) 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).

\( A_3 \) As long as \( Q \) holds, when a method of \( C \) is executed, the variant \( V \) must not increase.

\( A_4 \) 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.

\( A_5 \) 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 \( (F^\infty \text{pre}(M)) \) and \( G^\infty (\neg Q) \), therefore, for each of these hypotheses, we show respectively in Lemma 2 and Lemma 3 that, assuming that the code of the class is correct w.r.t. the annotations \( A_{1.5} \) \( (C : A_{1.5}) \), the satisfaction of \( Loop(Q,V,M) \) is established on \( \Sigma_C \).

**Lemma 2** If \( C : A_{1.5} \) and \( \sigma \in \Sigma_C \) and \( \sigma \models (F^\infty \text{pre}(M)) \) then \( \sigma \models Loop(Q,V,M) \).

**Lemma 3** If \( C : A_{1.5} \) and \( \sigma \in \Sigma_C \) and \( \sigma \models G^\infty (\neg Q) \) then \( \sigma \models Loop(Q,V,M) \).

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

**Theorem 1**

If \( C : A_{1.5} \) then \( \Sigma_{PH(Q,M)} \models Loop(Q,V,M) \).

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 \( \sigma \in \Sigma_C \), \( \sigma \models Loop(Q,V,M) \) and \( C : A_{1.5} \) 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.

   \[
   \text{after storeData called always perso;} \quad (S)
   \]

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.

   \[
   \text{after begin normal eventually !trDepth}
   \]

   under variant getBufferLess() for begin, commit, abort, write. \quad (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 \( \ldots \) for.

The result of the translation of Properties \( S \) and \( L \) is displayed in Fig. 3.
public class Buffer {
    //0 invariant witness_S = false; (S_a)
    //0 invariant witness_J = false; (L_a)
    //0 ghost boolean witness_S = false; (S_a)
    //0 ghost boolean witness_J = false; (L_a)
    //0 L ==> (L = !trDepth; L = false; L ==> (L = !trDepth; L = true; L = !trDepth; L = true)
    void storeData(int l) {
        //0 set witness_J = true; (S_b)
        //0 set witness_J = true; (L_c)
        void begin() {
            try { // L_c
                catch (Exception e) {
                throw e;
            }
        }
    }
    void write(byte b) {
        //0 set witness_J = !trDepth; (L_c)
    }
    void byte[] /*@ pure @*/ getStatus() {
    }
    void byte[] /*@ pure @*/ getStats() {
    }
}

void begin() {
    try { // L_c
        catch (Exception e) {
        throw e;
    }
}
    finally {
        //0 set witness_J = true;
    }
}

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

\[
\text{Loop}(\text{witness}_L, \text{getBufferLess}(), \{\text{begin, commit, abort, write}\})
\]

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

Notice that, since no method of Buffer diverges, Annotation 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 an 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