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

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

Following \([15]\), Property \(S\) is a safety property, expressing that \textit{something bad} – a state where \(¬P\) 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 modularity 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 \textit{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 \texttt{Loop}, inspired from \([9]\), for expressing liveness properties; (3) a way to verify on a class in isolation liveness properties expressed with the \texttt{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 \texttt{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 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 method write() 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 \texttt{old}. Using the clause \texttt{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 \texttt{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 \texttt{ensures} clause. A method may terminate exceptionally by throwing an exception and satisfying the exceptional post-condition (\texttt{signals} clause). The \texttt{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 \texttt{pure}. The specification of a method can also contain a \texttt{diverges} clause (not displayed in this example). If the predicate of a \texttt{diverges} clause of a method \texttt{m} is satisfied by the pre-state of \texttt{m}, then the execution of \texttt{m} may not terminate. Otherwise the method \texttt{must} terminate. By default, the JML \texttt{diverges} clause is set to \texttt{false}. A method with a \texttt{helper} modifier may not preserve the invariant. JML also introduces its own variables — declared with the keyword \texttt{ghost}. A special set annotation exists to assign their value.

In the rest of the paper, given a method \texttt{m}, we denote by \texttt{requires(m)} (resp. \texttt{diverges(m)}), the predicate of the \texttt{requires} clause of \texttt{m} (resp. the \texttt{diverges} clause).

The correctness of a Java class \texttt{C} w.r.t. a JML annotation \texttt{A}, denoted \texttt{C : 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 \textit{execution paths} the semantics of a class \texttt{C} in isolation. A class \texttt{C} is the description of a set of objects \texttt{I_C}, called the instances of \texttt{C}.

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

- \texttt{Cons_C \subseteq M_C} the set of constructors of the class.
- \texttt{Helper_C \subseteq M_C} the set of helper methods of the class.
- \texttt{PM_C} the set of progress (side-effect) methods. The set of non-progress (pure) methods is denoted by \texttt{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 \texttt{P} \in \texttt{Pred} be a JML predicate and let \texttt{s_{pre}} and \texttt{s_{cur}} be two memory states, we denote by \((\texttt{s_{pre}}, \texttt{s_{cur}}) \models \texttt{P}\) the satisfaction of \texttt{P} at these memory states. The values of the variables with an \texttt{old} in \texttt{s_{pre}} and the others in \texttt{s_{cur}}. If \texttt{P} does not contain variables relating to a preceding state, i.e., no \texttt{old}, then we simply denote \texttt{s_{cur}} \models \texttt{P}. Given a state \texttt{s} and a variable \texttt{a}, \texttt{s(a)} returns the value of \texttt{a} in the state \texttt{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 \texttt{s} be a memory state, assume to have the following special variables:

- \texttt{s(excp)}, a flag denoting that an exception has been thrown.
- \texttt{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;
    @pure void storeData(int l){
        @requires perso == false;
        @assignable len,perso;
        @normal_behavior
        @requires perso == false;
        @assignable buffer;
        @normal_behavior
        @requires trDepth == false;
        @assignable position;
        @assignable len;
        //© set trDepth = false;
        void begin() throws Exception{
            if (perso == false) {
                @pure byte [] getStatus(){
                    return status;
                }
                @pure int getBufferLess(){
                    return len - buffer.length;
                }
            }
            @normal_behavior
            @requires trDepth == false;
            @assignable buffer;
            @normal_behavior
            @requires trDepth == false;
            @assignable position;
            @assignable len;
            @assignable buffer;
            @assignable position;
            @assignable len;
            //© set trDepth = false;
        }
Let Σ be a set of paths, the visible state abstraction of Σ, denoted vsa_Σ(σ), be the set of instances of Σ, i.e.,

\[ \text{vsa}_Σ(Σ) = \{ \text{vsa}_i(σ) | σ ∈ Σ \}. \]

Then, we define the semantics of an instance ic, denoted vsa_ic(σ), by:

- vsa_ic(ε) = ε
- if \( s_0 \models \text{visible}(ic) \) then vsa_ic(σ) = s_0, vsa_ic(σ₁) else vsa_ic(σ) = vsa_ic(σ₁).

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:

\[ s_0[S]_{ic} = \text{def} \text{vsa}_ic(s_0[S]) \]

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 Σ be a set of paths, the visible state abstraction of Σ w.r.t. an instance ic, denoted vsa_Σ(ic) is defined as the set of all the abstractions of the paths of Σ, i.e.,

\[ \text{vsa}_Σ(Σ) = \{ \text{vsa}_i(σ) | σ ∈ Σ \}. \]

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

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

- \( ε ∈ Σ_{ic} \).
- Let \( s_0 \) be a state, let \( m ∈ \text{Cons}_C \). If \( s_0 \models \text{requires}(m) \) then
  \[ s_0[(m,ic)]_{ic} ∈ Σ_{ic}. \]
- Let \( σ ∈ Σ_{ic} \) be a finite execution and \( s_n \) be its last state. Let \( s_{n+1} \) be a state such that \( ∀v ∈ V_C.(s_n(v) = s_{n+1}(v)) \). Let \( m \) such that \( m ∈ M_C \land m \notin \text{Cons}_C \land m \notin \text{Helper}_C \). If \( s_{n+1} \models \text{requires}(m) \) then:
  \( (σ, (s_{n+1}[(m,ic)]_{ic})) ∈ Σ_{ic}. \)

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:

\[ Σ_C = \text{def} \bigcup_{ic ∈ I_C} Σ_{ic}. \]

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 \( Σ_C \) 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:

- 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. 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 (\( ∀a.(a ∈ V_S \land a \notin A \Rightarrow s_j(a) = s_j(a)) \)).

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.
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

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

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

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

Notice that the variant $V$ and the set of methods $M$ 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 $\sigma$ 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

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

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,

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

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 \( i \) such that any state of \( i \) satisfies \( \neg Q \).

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

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

\[
(G^\infty \neg Q) \lor (F^\infty \text{pre}(M))
\]

where \( \text{pre}(M) \) denotes the predicate \( \forall m \in M \; \text{pre}(m) \).

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

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

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

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

### 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 \( \text{Loop}(Q, V, M) \) be the Loop primitive. Let \( A_{1-5} \) be the following set of JML annotations.

\[
\begin{align*}
/A \text{ invariant } V & \geq 0; & (A_1) \\
/A \text{ constraint } Q & \Rightarrow V < \\text{old}(V) \text{ for } M; & (A_2) \\
/A \text{ constraint } Q & \Rightarrow V \leq \\text{old}(V); & (A_3) \\
/A \text{ invariant } Q & \Rightarrow \bigvee_{m \in M} \text{requires}(m); & (A_4) \\
/A \text{ invariant } Q & \Rightarrow \bigwedge_{m \in M} (\text{requires}(m) \Rightarrow !\text{diverges}(m)); & (A_5)
\end{align*}
\]

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 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 \( A_{1-5} \), the satisfaction of \( \text{Loop}(Q, V, M) \) 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 \( \sigma \models \text{Loop}(Q, V, M) \).

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

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

**Theorem 1**

If \( C : A_{1-5} \) then \( \Sigma_{C/PH(Q,M)} \models \text{Loop}(Q, V, M) \).

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

**Proposition 1** When \( M = \mathcal{P}M_C \), given \( \sigma \in \Sigma_C \), \( \sigma \models \text{Loop}(Q, V, M) \) 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.

   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 \( \text{trDepth} \) 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.
public class Buffer {

    // 0 ghost boolean witness \( S \) = false;  \( (S_o) \)
    // 0 ghost boolean witness \( L \) = false;  \( (L_o) \)
    // 0 invariant witness \( S \) 0 \( \Rightarrow \) perso; 0 +/
    // 0 invariant getBufferLess() > 0;  \( (S_i) \)
    // 0 constraint witness \( L \) \( \Rightarrow \)
        0 getBufferLess() \( \langle \! \langle \) \( \text{old}(\text{getBufferLess}) \) 0 for begin, comit, abort, write; 0 +/
    // 0 constraint witness \( L \) \( \Rightarrow \)
        0 getBufferLess() \( \rangle \rangle \langle \! \langle \) \( \text{old}(\text{getBufferLess}) \) 0 +/
    // 0 invariant witness \( L \) \( \Rightarrow \) (0 (trDepth = false \&\& perso = true ||
        0 (trDepth = true \&\& perso = true ||
            0, \&\& position < len))

    void storeData(int l){
        // 0 set witness \( S \) = true;  \( (S_i) \)
        // 0 set witness \( L \) = !trDepth;  \( (L_o) \)
        void begin(){
            try \( (L_o) \)
                // 0 set witness \( L \) = !trDepth;  \( (L_o) \)
                catch (Exception e) {
                    throw e;
                }
            finally {
                // 0 set witness \( L \) = true;
            }
        }
    }

    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 \( S_o \) in Fig. 3) and it is set to true when the method storeData is called (see annotation \( S_i \)). 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 \( L_o \) in Fig. 3). The ghost variable witness \( L \) is assigned using a try...catch...finally statement (see annotation \( L_o \)). 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 \( L_i \)).

    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 \( S_i \)). 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 \( L_i \)).

    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 a industrial Javacard application [3].

---

Figure 3: Buffer with generated annotations
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检查器, 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 assuming that a liveness is established on the class in isolation — the verification of properties expressed by Büchi automata.

An 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 advice.

7. REFERENCES


