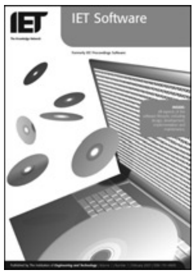
Published in IET Software
Received on 26th January 2008
Revised on 2nd July 2008
doi: 10.1049/iet-sen:20080008

In Special Issue on SAVCBS



ISSN 1751-8806

Verification of class liveness properties with Java modelling language

A. Giorgetti[1,2] *J. Groslambert*[3] *J. Julliand*[2]
O. Kouchnarenko[1,2]
[1]*INRIA/CASSIS*
[2]*LIFC/University of Franche-Comté, 16 route de Gray, Besançon Cedex F-25030, France*
[3]*Trusted Labs, 5 rue du Bailliage, Versailles F-78000, France*
E-mail: kouchna@lifc.univ-fcomte.fr

Abstract: Static checking is key for the security of software components. As a component model, this paper considers a Java class enriched with annotations from the Java modelling language (JML). It defines a formal execution semantics for repetitive method invocations from this annotated class, called the class in isolation semantics. Afterwards, a pattern of liveness properties is defined, together with its formal semantics, providing a foundation for both static and runtime checking. This pattern is then inscribed in a complete language of temporal properties, called Java temporal pattern language, extending JML. The authors 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 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

Component-based development provides significant advantages – portability, adaptability, re-usability and so on – when developing, for example Java Card smart card applications [1] or when composing Web services within service component architecture (SCA) – a relatively new initiative advocated by users of Java technology. In this framework, the use of components of distributed applications or component-based applications necessitates ensuring not only invariance and safety properties but also partial correctness and liveness properties of components. We consider a component modelled by a Java class that is annotated in Java modelling language (JML).

Currently, more and more tools aiming at the verification of Java programs are adopting JML as property specification language (see [2] for an overview). JML is a specification language syntactically and semantically close to Java, thus making specifications more accessible to Java programmers (see http://www.jmlspecs.org). JML allows adding basic formal annotations – like method pre- and post-conditions or invariants – to the Java class, thus proposing a way to modularly verify Java applications. However, it is difficult to directly specify complex dynamic properties in JML, like temporal properties [3], that are often needed to express the security policies that the Java implementation has to ensure. Therefore Huisman and Trentelman [4] proposed a language of temporal properties – later called JTPL, for Java temporal pattern language [5].

Our main purpose is to verify liveness properties of Java/JML components using a JML extension. The first contribution is a formal execution semantics for repetitive method invocations from this component, called the class in isolation semantics. To infer class invariants by abstract interpretation, Logozzo [6] proposed a semantics of partial execution paths of an object-oriented program and of the so-called class in isolation. Our work follows this approach but, for verifying liveness properties, we define a complete execution path semantics of a class in isolation. Moreover, since we consider Java/JML components, we take into account the JML semantics to define the class in isolation semantics. The second contribution is an extension of the JML type specifications with a temporal specification of liveness by introducing a new specification clause in Java classes – called the liveness clause. A deep integration of this liveness clause in JML is achieved by using the same semantics of visible states as for JML invariant and constraint clauses. The third contribution is a verification method for liveness properties by generating JML invariants and history constraints. The fourth contribution is a systematic translation of temporal properties into JML annotations. Thanks to the semantics, we establish the correctness of the translation. Notice that the second and third contributions were presented in [7] without proofs nor explicit examples. To make it short, the main extensions to [7] are (1) a complete formal semantics for the executions of a Java/JML component, and (2) the translation of all the liveness formulae of the JTPL temporal logic into standard JML annotations.

This paper is organised as follows: Section 2 quickly presents JML on an example. Section 3 introduces the mathematical background used in the next sections. Section 4 presents the semantic framework of the paper. In particular, it defines a semantics for a class in isolation and gives the semantics of JML main annotations. Moreover, the semantics of visible states is recalled and formalised (upon an *ad hoc* semantics of a Java class). Next, Section 5 defines the liveness clause and its formal semantics. Section 5 also presents the verification of liveness properties on a class in isolation through appropriate annotation generation. Section 6 presents the application of the annotation generation method to the JTPL temporal liveness properties based on their translation into the liveness clause that we propose to extend JML. Section 7 presents the JAG (JML annotation generator) tool implementing this automatic generation of annotations. Section 8 concludes by giving some perspectives and future work.

2 Overview of JML and example

JML [8] is a specification language especially tailored for Java applications. Originally, JML was proposed by 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 more especially to specify smart card applications [1, 9]. JML is developed following the design by contract approach [10], where classes are annotated with class invariants and method pre- and post-conditions. The predicates are side-effect free Boolean Java expressions, extended with specific constructs. Specifications are written as Java comments marked with an @, that is, annotations follow `//@` or are enclosed between `/*@` and `@*/`. Fig. 1 presents some JML annotations on the simple example of a buffer.

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

Fig. 1 displays a class `invariant`, that is, a predicate that has to hold on every so-called JML visible state. 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 methods list for which the history constraint must be satisfied. When this clause is omitted, the constraint must hold for all the class methods. The clause `requires` denotes the pre-condition of the method, that is, a predicate that must be true when the method is called. A post-condition is expressed with an `ensures` clause. A method may exceptionally terminate by throwing an exception and satisfying the exceptional post-condition (`signals` clause). The method specification 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`. JML also introduces its own variables - declared with the keyword `ghost`. A special `set` annotation exists to assign their value. For instance, `trDepth = true` means that a transaction is in progress. This variable allows expressing that every opened transaction must eventually be closed. This is an example of liveness property that will be translated into a set of JML annotations. The correctness of a Java class w.r.t. JML annotations can be established by model-checking [11] or by a prover (B or Coq) via a proof obligation generator (Jack [12] or Krakatoa [13]).

3 Preliminaries

This section introduces some definitions and notations used in the other sections. It recalls the notion of sequence and some useful results for the existence of fixpoints in lattices.

3.1 Notations

Familiarity with basic set theory is assumed. Given a binary relation $R \subseteq S_1 \times S_2$, dom($R$) is its domain, ran($R$) is its range and R^{-1} is the inverse relation. If dom(R) $= S_1$, then the relation is total. A relation $f \subseteq S_1 \times S_2$ is a partial function from S_1 to S_2, if each element of its domain has a single image. It is a (total) function, denoted $S_1 \rightarrow S_2$, if it is total and a partial function. An endofunction of S is a function from S to itself. For any function f: $S_1 \rightarrow S_2$, $x \in S_1$ and $y \in S_2$, the update of f with y at x, denoted

```
public class Buffer {
  private int len;
  private byte[] status;
  private byte[] buffer;
  private int position = 0;
  private boolean customized = false;

//@ ghost boolean trDepth = false;

//@ invariant position >= 0;

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

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

byte[] getStatus(){
  return status;
}
```

```
int getBufferLess(){
  return len - buffer.length;
}

/*@ normal_behavior
  @   requires trDepth == false;
  @   requires customized == true;
  @ also
  @ exceptional_behavior
  @   requires customized == false;
  @   signals (Exception e) true;
  @*/
void begin() throws Exception{
  if (customized == false) {
    throw new Exception();
  }
  buffer = new byte[len];
  //@ set trDepth = true;
}

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

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

/*@ normal_behavior
  @ requires trDepth == true;
  @ requires customized == true;
  @ requires position
  @   + b.length <= len;
  @ diverges false;
  @ ensures position <= len;
  @ ensures position ==
  @   \old(position)+b.length;
  @*/
void write(byte[] b){
  int i = 0;
  while (i < b.length){
    buffer[position] = b[i];
    position++;
    i++;
  }
}
}
```

Figure 1 *JML annotated transaction system*
Every opened transaction must eventually be closed

$f[x \mapsto y]$, is the unique function such that

$$f[x \mapsto y](u) = \begin{cases} f(u) & \text{if } u \neq x \\ y & \text{if } u = x \end{cases}$$

More generally, we write $f[x_1 \mapsto y_1, \ldots, x_n \mapsto y_n]$, instead of $f[x_1 \mapsto y_1] \ldots [x_n \mapsto y_n]$, when $x_1 \ldots x_n$ are all different.

3.2 Sequences

Let S be a (non-empty) set. A sequence is a partial function σ from $\mathbb{N}$ to S such that the set $\mathrm{dom}(\sigma)$ is either $\mathbb{N}$ or a finite subset $[0, \ldots, k]$ for some k in $\mathbb{N}$. The empty sequence, whose domain is the empty set, is denoted ϵ. A sequence σ is infinite if $\mathrm{dom}(\sigma) = \mathbb{N}$, finite otherwise. The length $\mathrm{len}(\sigma)$ of a sequence σ is n if it is finite and if $\mathrm{dom}(\sigma) = [0, \ldots, n-1]$, ω otherwise. The last element $\mathrm{last}(\sigma)$ of a sequence σ is $\sigma(\mathrm{len}(\sigma) - 1)$ if this sequence is finite and non-empty, ω otherwise. We use S^*, S^+, S^ω, $S^{*\omega}$ and $S^{+\omega}$ to, respectively, denote the sets of finite, non-empty finite, infinite, finite or infinite, and non-empty finite or infinite sequences.

The concatenation $\alpha \cdot \beta$ of two sequences $\alpha, \beta \in S^{*\omega}$ of length $l = \mathrm{len}(\alpha)$ and $m = \mathrm{len}(\beta)$ is a sequence of length $\mathrm{len}(\alpha \cdot \beta) = l \oplus m$, where $\oplus$ extends the addition of $\mathbb{N}$ to $\mathbb{N} \cup \{\omega\}$, with $\omega \oplus n = n \oplus \omega = \omega$ for any $n \in \mathbb{N} \cup \{\omega\}$. $\sigma = \alpha \cdot \beta$ is defined by $\sigma(i) = \alpha(i)$ for $0 \leq i < l$ and $\sigma(i) = \beta(i - l)$ for $l \leq i < \mathrm{len}(\sigma)$. Concatenation of sequences extends to sets of sequences in a standard way, with the same notation.

Two non-empty sequences $\alpha, \beta \in S^{+\omega}$ of length $l = \mathrm{len}(\alpha)$ and $m = \mathrm{len}(\beta)$ are joinable iff $\mathrm{last}(\alpha)$ is $\beta(0)$ or ω. When they are joinable, their join (or junction) $\alpha \frown \beta$ is a sequence σ of length $\mathrm{len}(\sigma) = (l \oplus m) \ominus 1$, where $\ominus$ extends the subtraction of $\mathbb{N}$ to $\mathbb{N} \cup \{\omega\}$, with $\omega \ominus n = \omega$ for any $n \in \mathbb{N} \cup \{\omega\}$. $\sigma = \alpha \frown \beta$ is such that $\sigma(i) = \alpha(i)$ for all $0 \leq i < l$ and $\sigma(i) = \beta(i - l + 1)$ for all $l \leq i < \mathrm{len}(\sigma)$ when $l < \omega$. The junction $S \frown T$ of the sets of non-empty sequences S and T is the set of junctions $\alpha \frown \beta$ of joinable sequences $\alpha \in S$ and $\beta \in T$.

3.3 Complete lattices

A partial order $\sqsubseteq$ on a set S is a relation on S which is reflexive ($\forall x \in S \cdot x \sqsubseteq x$), transitive ($\forall x, y, z \in S \cdot (x \sqsubseteq y \wedge y \sqsubseteq z) \Rightarrow x \sqsubseteq z$) and antisymmetric ($\forall x, y \in S \cdot (x \sqsubseteq y \wedge y \sqsubseteq x) \Rightarrow x = y$). A partially ordered set $\langle S, \sqsubseteq \rangle$, or poset, is a set equipped with a partial order $\sqsubseteq$. A lower bound l of $U \subseteq S$ is an element l of S such that $\forall x \in U \cdot l \sqsubseteq x$. A greatest lower bound of U is a lower bound g of U such that $l \sqsubseteq g$ holds for all lower bound l of U. A (least) upper bound of U for $\sqsubseteq$ is a (greatest) lower bound of U for the inverse partial order $\sqsubseteq^{-1}$. By antisymmetry of $\sqsubseteq$, greatest lower and least upper bounds, when they exist, are unique.

A complete lattice $\langle S, \sqsubseteq, \sqcup, \sqcap \rangle$ is a poset $\langle S, \sqsubseteq \rangle$ where every subset $U \subseteq S$ has a least upper bound, denoted $\sqcap U$, and a greatest lower bound, denoted $\sqcup U$. An endofunction f of S is monotone if $\forall x, y \in S \cdot x \sqsubseteq y \Rightarrow f(x) \sqsubseteq f(y)$. A consequence of Tarski's fixpoint theorem [14] is the existence of least and greatest fixpoints for any monotone function in a complete lattice.

Proposition 1: Every monotone endofunction f on a complete lattice $\langle L, \sqsubseteq, \sqcup, \sqcap \rangle$ admits a least fixpoint $\mathrm{lfp}(f) =_{\mathtt{def}} \sqcap \{x | x \in L \wedge f(x) \sqsubseteq x\}$ and a greatest fixpoint $\mathrm{gfp}(f) =_{\mathtt{def}} \sqcup \{x | x \in L \wedge x \sqsubseteq f(x)\}$.

3.4 Sequence set lattice

When a program can either run forever or end, its execution (or trace) semantics is a set of finite or infinite sequences (of states). Following [15], these sets can be specified as fixpoints in the set $2^{S^{+\omega}}$ of sets of non-empty finite or infinite sequences. The following proposition defines a lattice over this set by fusion of the complete lattices $\langle 2^{S^{+}}, \subseteq, \cup, \cap \rangle$ and $\langle 2^{S^{\omega}}, \supseteq, \cap, \cup \rangle$ of sets of, respectively, non-empty finite and infinite sequences. A proof that they are complete lattices can be found in [10], Th. 11 and 12. In all that follows, X^{+} (resp. X^{ω}) shortens $X \cap S^{+}$ (resp. $X \cap S^{\omega}$) for any X in $2^{S^{+\omega}}$.

Proposition 2: (Corollary of [15], Theorem 9): Let $2^{S^{+\omega}}$ be the (disjoint) union of $2^{S^{+}}$ and $2^{S^{\omega}}$. For any X in $2^{S^{+\omega}}$, let $\sqsubseteq$ be defined by $X \sqsubseteq Y = X^{+} \subseteq Y^{+} \wedge X^{\omega} \supseteq Y^{\omega}$. For any subset Z of $2^{S^{+\omega}}$, let $\sqcup$ and $\sqcap$ be, respectively, defined by $\sqcup Z = \bigcup_{X \in Z} X^{+} \cup \bigcap_{X \in Z} X^{\omega}$ and $\sqcap Z = \bigcap_{X \in Z} X^{+} \cup \bigcup_{X \in Z} X^{\omega}$.

Then $\langle 2^{S^{+\omega}}, \sqsubseteq, \sqcup, \sqcap \rangle$ is a complete lattice.

4 Execution semantics

Our aim is to verify liveness properties of Java/JML components. A suitable semantics for this is a set of maximal execution paths. Intuitively, an execution path (or simply an execution) is a sequence of states reached during an execution of the class. An execution path is maximal if it cannot be extended to form a longer execution path. A maximal execution path is either infinite or is terminating with a blocking state.

4.1 Context restrictions

We study a component that is a Java class enriched with some JML annotations: `invariant`, `constraint` and `ghost` variables for the class, `behavior` for methods and `set` in their bodies. The annotation `pure` means that a method is side-effect free. The annotations `helper` and `assignable` are useless in defining the liveness properties that we address. Consequently, we do not take these annotations into account. We do not address the problems of inheritance, multithreading and exception hierarchy. To simplify the presentation, we do not take into account the finalisers and the static methods. The execution of the component environment is restricted to creating only one instance of the class. The execution invokes only the non-static methods.

We assume that the environment and the class respect the contract defined by the JML specifications. That means that the environment calls method m from a memory state that satisfies its `requires` condition. It is assumed that the annotated class is consistent, that is, each method m leads to a state that satisfies either its `ensures` condition if m does not diverge and does not raise an exception or the `signals` predicate if it raises an exception.

4.2 Java subset semantics

In this section, Definitions 1 and 2 describe the Java/JML components that we consider. Then an execution semantics of a Java subset is given in Definition 4 as a sequence of memory states, defined in Definition 3.

A component is a Java class defining a set of methods and a set of attributes and ghost variables. The class can be annotated with JML annotations as `invariant`, `constraint` and `behavior`. A behaviour is a method annotation. A class can also contain `ghost` variables and `set` annotations. A component is, therefore, an annotated class in Java/JML defined as follows:

Definition 1 (annotated class): An annotated class C is a tuple (V_C, I_C, C_C, M_C) where V_C is the set of attributes and ghost variables of the class, I_C is a set of JML invariants, C_C is a set of JML constraints and M_C is the set of all method names of the class except the constructor i_C. A method named m in M_C is defined by a tuple $(\mathrm{behaviour}_m, \mathrm{paramList}_m, \mathrm{body}_m)$ where $\mathrm{behaviour}_m$ is the JML specification of a canonical behaviour, $\mathrm{paramList}_m$ is its set of parameters and body_m is the Java program that implements m.

By a desugaring operation [16], the method m behaviours can be reduced to a single canonical behaviour annotation: `behavior; requires` P_m`; diverges` D_m`; ensures` Q_m`; signals (Exception e)` R_m`;`. In the rest of the paper, P_m, D_m, Q_m and R_m, respectively, denote the requires, diverges, ensures and signals predicates of the behaviour of method m.

The addressed Java subset to define method bodies body_m is composed of atomic and method call statements, respectively, denoted by as and $m(E, \ldots, E)$, sequential, conditional and iterative statement compositions, and exception handling. An atomic statement is any statement that does not define other memory states than the states before and after its execution. A typical example is an assignment of a variable in V_C.

Definition 2 (Java subset): Let E be a Java expression, m a Java identifier and P a Java predicate (a Boolean Java expression). We consider the Java statement subset T defined by the following abstract syntax:

$$\begin{aligned} T ::= &\ as \mid m(E, \ldots, E) \mid T\ ;\ T \mid \mathtt{if}\ (P)\ \{T\}\ \mathtt{else}\ \{T\} \\ &\mid \mathtt{while}\ (P)\ \{T\} \mid \mathtt{throw} \mid \mathtt{try}\ \{T\}\ \mathtt{catch}\ \{T\} \\ &\mid \mathtt{try}\ \{T\}\ \mathtt{finally}\ \{T\}. \end{aligned}$$

A memory state assigns values to variables. For a component in isolation, we consider three sets of variables, namely: the set V_C of attributes and ghost variables, the set $P_C = \cup_{m \in M_C}$ paramList$_m$ of parameters of all the methods (to simplify, we assume that distinct methods have disjoint parameter sets), and a set of three special variables to control the execution.

Definition 3 (memory state): A memory state s is composed of:

- two total functions $V_C \rightarrow$ VAL and $P_C \rightarrow$ VAL $\cup \{\perp\}$, where VAL is the set of all values of the different Java types and $\perp \notin$ VAL; the former function assigns a value to any attribute and ghost variable of C; the latter assigns a value to any parameter of any method of C; when the parameter is not used, its value is undefined – denoted $\perp$.

- a boolean variable excp; the predicate s(excp) indicates that an exception has been thrown,

- a variable $cM \in M_C$, indicating the name of the method currently performed,

- a variable sH, that is a natural number that represents the height of the execution stack.

This definition simplifies memory models for object-oriented languages [13, 17]. The Java memory also contains an execution stack [18]. As in[6], we do not explicitly use the execution stack, but we observe it with the threespecial variables excp, cM and sH. Given a state s and a variable x in $V_C \cup P_C \cup \{$excp, cM, $sH\}$, $s(x)$ denotes the value of x in the state s, when it is defined. We denote by STATE the set of memory states of a class C.

Let E be a Java/JML expression. We consider an evaluation function, written eval(E, s), that returns the value of E in the state s. We suppose that expressions are side-effect free and do not contain method calls. They are denoted by E, E_i. JML predicates are boolean expressions defined over attributes, ghost variables and values of their types. Some predicates, for example in the `constraint` or `ensures` clauses, are pre-/post-predicates using the values of variables in the previous state by the `\old` notation. Let P be a JML predicate and s, s' be two memory states. If P does not contain the keyword `\old`, $s \models P$ denotes that eval(E, s) = `true`. Otherwise, $(s, s') \models P$ denotes that the evaluation of P w.r.t. the states s and s' is true. The subterms t of P appearing as `\old(t)` in P are evaluated in the state s, and the subterms t' that are not included in the keyword `\old` are evaluated in the state s'.

Intuitively, we define the semantics of a Java statement T as the execution $s[\![T]\!]$ that is generated by the execution of T from the memory state s of STATE. An execution is a sequence of memory states, that is, an element of STATE$^{+\omega}$.

Definition 4 (Java subset semantics): Let T be a Java statement and $s \in$ STATE a memory state without exception ($\neg s(excp)$). The execution $s[\![T]\!]$ is defined in Fig. 2, where f_{as} : STATE $\rightarrow$ STATE is the state transformer of the atomic statement as.

Each equality in this definition must be understood as follows:

1. The execution for an atomic statement *as* is the output state resulting from *as*.

2. When a method m is called, the first state s_{in} is a pre-state that contains the value of every parameter of m. In this state, the current method is m and the stack height is incremented. If s_{in} does not satisfy the pre-condition P_m of method m, the

$$s[\![as]\!] = f_{as}(s)$$

$$\begin{array}{l}
s[\![m(E_1,\ldots,E_n)]\!] = \text{ let } s_{in} = s[p_1 \mapsto eval(E_1,s),\ldots,p_n \mapsto eval(E_n,s), cM \mapsto m, sH \mapsto s(sH)+1] \\
\quad \text{in if } s_{in} \models \neg P_m \text{ then } s[excp \mapsto true] \text{ else} \\
\quad \text{let } \sigma = s_{in}[\![body_m]\!] \text{ in if } len(\sigma) = \omega \text{ then } s_{in}.\sigma \text{ else} \\
\quad \text{let } s_{exit} = s[excp \mapsto last(\sigma)(excp), \{a \mapsto v \mid a \in V_C \wedge v = last(\sigma)(a)\}] \text{ in } s_{in}.\sigma.s_{exit}
\end{array}$$

$$\begin{array}{l}
s[\![T_1;T_2]\!] = \text{ let } \sigma = s[\![T_1]\!] \text{ in if } len(\sigma) = \omega \text{ then } \sigma \text{ else} \\
\quad \text{let } s' = last(\sigma) \text{ in if } s'(excp) \text{ then } \sigma \text{ else } \sigma.s'[\![T_2]\!]
\end{array}$$

$$s[\![\texttt{if } (P)\{T_1\} \texttt{ else } \{T_2\}]\!] = \text{ if } s \models P \text{ then } s[\![T_1]\!] \text{ else } s[\![T_2]\!]$$

$$\begin{array}{l}
s[\![\texttt{while } (P)\{T\}]\!] = \text{ if } s \models \neg P \text{ then } s \text{ else} \\
\quad \text{let } \sigma = s[\![T]\!] \text{ in if } len(\sigma) = \omega \text{ then } \sigma \text{ else} \\
\quad \text{let } s' = last(\sigma) \text{ in if } s'(excp) \text{ then } \sigma \text{ else } \sigma.s'[\![\texttt{while}(P)\{T\}]\!]
\end{array}$$

$$\begin{array}{l}
s[\![\texttt{try } \{T_1\} \texttt{ catch } \{T_2\}]\!] = \text{ let } \sigma = s[\![T_1]\!] \\
\quad \text{in if } len(\sigma) = \omega \text{ then } \sigma \text{ else} \\
\quad \text{let } s' = last(\sigma) \text{ in if } \neg s'(excp) \text{ then } \sigma \text{ else } \sigma.(s'[excp \mapsto false])[\![T_2]\!]
\end{array}$$

$$\begin{array}{l}
s[\![\texttt{try } \{T_1\} \texttt{ finally } \{T_2\}]\!] = \text{ let } \sigma = s[\![T_1]\!] \\
\quad \text{in if } len(\sigma) = \omega \text{ then } \sigma \text{ else} \\
\quad \text{let } s' = last(\sigma) \text{ in } \sigma.(s'[excp \mapsto false])[\![T_2]\!]
\end{array}$$

$$s[\![\texttt{throw}]\!] = s[excp \mapsto true]$$

Figure 2 *Java subset semantics*

execution raises an exception. Otherwise, the state s_{in} is followed by the sequence of states resulting from the execution of the body of m. When this execution is finite, it ends with a last state s_{exit} whose parameters, cM and sH, are equal to their values in the first state s and whose other values are those of the last state of the body execution.

3. The execution for a sequence of T_1 and T_2 is the concatenation of the executions of T_1 and T_2 if T_1 terminates without raising an exception. Otherwise, it is the execution of T_1.

4. The execution for the conditional statement is the execution of T_1 if s satisfies P and the execution of T_2 otherwise.

5. Defining the execution semantics of the iterative `while` statement is a difficult point. For any predicate P and statement T, this execution is either reduced to the sequence s if the state s does not satisfy P, or the execution of T if T does not terminate or raises an exception, or the concatenation of a first execution of T and the execution of the same iterative statement from the last state of this first execution of T. This is the intended meaning of the fifth equality in Fig. 2. The trouble is that this 'definition' of ($\lambda s.s[\![\texttt{while}\ (P)\{T\}]\!]$) is circular. The question remains whether this equation admits a solution and, if it admits more than one, which one should be retained as the right definition. A basic answer is to define this solution as a fixpoint over an adequate lattice. Consider the set $[\![T]\!]$ of executions starting from any memory state in STATE. This set is related to the unique execution $s[\![T]\!]$ after a given state s by $[\![T]\!] = \{s.\sigma | s \in \text{STATE} \wedge \sigma \in s[\![T]\!]\}$. $[\![\texttt{while}\ (P)\{T\}]\!]$ could be defined in the sequence set lattice from Proposition 2 as the least fixpoint of the endofunction W defined by $W = (\lambda X \cdot \{\sigma \in \text{STATE}^{+\omega} \mid \text{len}(\sigma) = 1 \wedge \sigma(0) \not\models P\} \cup \{\sigma \in \text{STATE}^{+\omega} | \sigma \in [\![T]\!] \wedge \sigma(0) \models P \wedge \text{last}(\sigma)(\text{excp})\} \cup \{\sigma \in \text{STATE}^{+\omega} | \sigma \in [\![T]\!] \wedge \sigma(0) \models P \wedge \neg\text{last}(\sigma)(\text{excp})\}^\frown X)$ with the convention that $\omega(\text{excp}) = \text{false}$. On the one hand, this function is monotone. On the other hand, a proof by induction on the statement language shows that the execution set $W(X)$ contains exactly one execution starting from any given state, for any execution set X. Then $s[\![\texttt{while}\ (P)\ \{T\}]\!]$ is defined as the execution starting with s in the least fixpoint of W.

6. The execution for the `try` $\{T_1\}$ `catch` $\{T_2\}$ statement is the execution of T_1 if T_1 either does not terminate or terminates without raising an exception. Otherwise, when T_1 terminates, the raised exception is removed and the execution continues with the execution of T_2.

7. The execution for the `try` $\{T_1\}$ `finally` $\{T_2\}$ statement is the execution of T_1 if T_1 does not terminate. Otherwise, when T_1 terminates either normally or by throwing an exception, the raised exception is caught if necessary and the execution continues with the execution of T_2.

8. The `throw` statement assigns the special variable excp to true. If the `throw` statement is in the T_1 part of a `try`$\{T_1\}$`catch`$\{T_2\}$ statement, the execution continues with the execution of T_2 as it is specified by its semantics. Otherwise, the execution stops.

One might expect the same semantics for $s[\![\texttt{while}\ (P)\ \{T\}]\!]$ and $s[\![m()]\!]$ when $m()$ is for instance defined by `void` $m()$ `{if` (P) $\{T;\ m()\}$ `else {...}}` where the dots are replaced with some statement producing an empty execution. In fact, these two semantics differ for the following reasons. The main reason is that the semantics of every method call has to begin with an entry state s_{in} and, if finite, to end with an exit state s_{exit}, whereas these two states should not appear in the semantics of the `while` statement. These two states for method calls are compulsory because the execution semantics is designed for defining the `ensures` JML clause in Section 4.6, that must hold between this entry state and the corresponding exit state. Another reason is that a method call assigns the special variables cM and sH, whereas a `while` statement does not modify them. Modulo these differences, $s[\![\texttt{while}(P)\ \{T\}]\!]$ is solution of an equation obtainable from the equation satisfied by $s[\![m()]\!]$ by removing the first state of each computed sequence and also the last state of each finite one.

4.3 Class semantics

As explained in Section 1, we aim to verify that a class C satisfies a liveness property. This satisfaction obviously depends on the context of use of that class. Here, we focus on the life cycle of a single object of type C, after its construction. We assume the encapsulation hypothesis, that is, that the class attributes can be modified only by the invocation of class methods. Consequently, the class use only depends upon the manner invoking the class methods. The class executions result from the activation of the constructor followed by a finite or infinite sequence of method calls that respect the contract – each of them protected by an exception recuperation statement. This class semantics $\Sigma_C^{+\omega}$ is defined in this section.

A method execution at toplevel is a (maximal) execution of a method m that starts from any state where the execution stack is empty and the exception flag is down. The set of executions of a class C at toplevel is denoted by $[\![C]\!]$ and defined by

$$\begin{aligned}[\![C]\!] =_{\text{def}} \{s.s[\![\texttt{try}\{m(v(p_1), \ldots, v(p_n))\}\texttt{catch}\{\}]\!] \mid m \in M_C \wedge \\ v \in \text{paramList}_m \to \text{VAL} \wedge s \in \text{STATE} \wedge \neg s(\text{excp}) \\ \wedge\ s(sH) = 0 \wedge s \models P_m\}\end{aligned}$$

where $\text{paramList}_m = \{p_1, \ldots, p_n\}$.

Let C be an annotated class, STATE its set of states, $[\![C]\!] \in \text{STATE}^{+\omega}$ its execution semantics and $S_0 \subseteq$ STATE the set of initial states resulting from the constructor i_C of C. The set

$f(C)$ of blocking (or final) states for the class C is defined by $f(C) = \mathrm{STATE}\backslash\{\sigma(0) \mid \sigma \in [\![C]\!]\}$. With these notations, the maximal execution semantics of an annotated class can be defined thanks to the following endofunction.

Proposition 3: In the complete lattice $\langle 2^{\mathrm{STATE}^{+\omega}}, \sqsubseteq, \sqcup, \sqcap \rangle$, the endofunction F defined by $F(X) = f(C) \cup ([\![C]\!]^\frown X)$ is monotone.

Proof: Notations are the same as in Proposition 2, except for S replaced here with the set STATE of memory states. By separating finite and infinite sequences, one has $F(X)^+ = f(C) \cup ([\![C]\!]^{+\frown} X^+)$ and $F(X)^\omega = ([\![C]\!]^{+\frown} X^\omega) \cup [\![C]\!]^\omega$. $X \sqsubseteq Y$ implies that $F(X)^+ \subseteq F(Y)^+$ and $F(X)^\omega \supseteq F(Y)^\omega$, that is, $F(X) \sqsubseteq F(Y)$. □

When $[\![C]\!]$ is a transition relation, that is, when it is a set of executions of length 2, this proposition is a corollary of Theorem 13 from [15], proved by fusion of fixpoints on the two lattices of non-empty finite and infinite executions. The present result is more general, since $[\![C]\!]$ may contain finite executions of any length, and even infinite executions. A consequence is that a proof by fusion is no more possible.

By Propositions 1 and 3, F admits a leastfixpoint, denoted by $\mathrm{lfp}(F)$.

Definition 5 (class semantics): The restriction of $\mathrm{lfp}(F)$ to executions starting from the states resulting from the constructor is called the class semantics and is denoted by $\Sigma_C^{+\omega} =_{\mathrm{def}} \mathrm{lfp}(F) \cap (S_0{}^\frown \mathrm{STATE}^{+\omega})$.

4.4 Visible states

The semantics of the JML `invariant` and `constraint` clauses is based on the notion of 'visible' states. This section formalises this notion and its semantics. Under the hypotheses of Section 4.1, the original definition of visible states, given in the JML reference manual [8], is restricted to three cases, as follows. A visible state is a state that occurs at one of these moments in a program's execution: at the first state of the execution, just after the end of a constructor invocation that has created the executed object; at the beginning or end of a (non-static non-finaliser) method invocation; outside of the execution of any constructor, finaliser or method when the execution stack is empty.

Let us first formalise the notions of pre- and post-states for a method m as follows.

Definition 6 (pre- and post-states): Let C be a class, $\sigma \in \Sigma_C^{+\omega}$ an execution, m a method of class C and $0 \le i < \mathrm{len}(\sigma)$. For $i > 0$, the ith state $\sigma(i)$ of σ is a pre-state of m, denoted prestate(σ, i, m), if $\sigma(i)(cM) = m$ and $\sigma(i)(sH) = \sigma(i-1)(sH) + 1$. The ith state of σ is a post-state of m, denoted poststate(σ, i, m), if $\sigma(i)(cM) = m$ and $\sigma(i)(sH) = \sigma(i+1)(sH) + 1$.

With this definition, for any execution of class C, we formalise – in conformity with [8] – what a visible state is.

Definition 7 (visible states): Given an execution $\sigma \in \Sigma_C^{+\omega}$, the ith state $\sigma(i)$ of σ is a visible state, denoted visible(σ, i), iff $i = 0$, $\sigma(i)(sH) = 0$ or there is a method $m \in M_C$ in C s.t. pre-state(σ, i, m) or post-state(σ, i, m).

It is now possible to abstract any execution by keeping only its visible states. The following definition of this abstraction is based on an auxiliary partial function $nv : \mathbb{N} \times \Sigma_C^{+\omega} \to \mathbb{N}$, such that $nv(i, \sigma)$ is the position of the $i + 1$th visible state in σ, when it exists. Let $\min(S)$ denote the minimum of any subset S of $\mathbb{N}$. nv is inductively defined by $nv(0, \sigma) = \min(\{j \mid 0 \le j < \mathrm{len}(\sigma) \wedge \mathrm{visible}(\sigma, j)\})$ and $nv(i, \sigma) = \min(\{j \mid nv(i-1, \sigma) < j < \mathrm{len}(\sigma) \wedge \mathrm{visible}(\sigma, j)\})$ for $i > 0$.

Definition 8 (visible state abstraction): The visible state abstraction of a class C, denoted vsa_C, is the endofunction of $\Sigma_C^{+\omega}$ defined by $\mathrm{vsa}_C(\sigma)(i) = \sigma(nv(i, \sigma))$ for any σ in $\Sigma_C^{+\omega}$ and any $0 \le i < \mathrm{len}(\sigma)$.

4.5 Class in isolation semantics

The semantics of a class in isolation is defined as the set of abstractions to visible states of complete (maximal) class executions. Following [6], this execution semantics is called the class in isolation semantics. It is defined as follows:

Definition 9 (class in isolation semantics): The class in isolation semantics of a class C is defined by $\Sigma_C =_{\mathtt{def}} \{\mathrm{vsa}_C(\sigma) \mid \sigma \in \Sigma_C^{+\omega}\}$.

4.6 Annotated class consistency

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 e.g. [13]), but never in terms of properties of the executions. In this section, we give an execution semantics of JML annotations defining their consistency with the set of executions Σ_C of the class in isolation.

In an annotated class, there are three canonical kinds of annotations: `invariant`, `constraint` and `behavior`. Their semantics are given by Definition 11 w.r.t. the definition in [8]. In Definition 11, we use the predicate $mp(\sigma, j, m, i)$ that is true if $\sigma(j)$ is the matching post-state of the pre-state $\sigma(i)$ (Definition 10).

Definition 10 (matching post-state of a state for a method in an execution): The jth state of $\sigma \in \Sigma_C^{+\omega}$ is the matching post-state of the ith state of σ for method m, denoted $mp(\sigma, j, m, i)$, if

$$\mathrm{poststate}(\sigma, j, m) \wedge \sigma(j)(sH) = \sigma(i)(sH) \wedge \forall k.$$
$$(i < k < j \Rightarrow \sigma(k)(sH) \ge \sigma(i)(sH)).$$

Definition 11 (consistency): Let C be an annotated class. We define that an execution σ of Σ_C satisfies a JML annotation $\mathcal{A}$ of the class C, denoted $\sigma:\mathcal{A}$, according to the formulae in Fig. 3.

This definition must be understood as follows:

• `Invariant`: The invariant must be satisfied by each visible state [see (1) in Fig. 3].

• `Constraint`: For the body of each method included in the `for` clause, the constraint must hold between two consecutive visible states that arise during the execution of the method, that is, all visible states between the pre-state and the matching post-state of the method [see (2) in Fig. 3].

• `Behavior` method specification: This JML specification is interpreted over an execution as follows. If the predicate P_m of the `requires` clause is satisfied by the pre-state of the method m, that implies:

– If D_m does not hold ($\neg D_m$), then the method must terminate, that is, it must have a post-state. Moreover, if it is a normal termination ($\sigma(j)$(excp)), the predicate Q_m of the `ensures` clause must be satisfied between the pre-state and the post-state, and the predicate R_m of the `signals` clause must be satisfied otherwise (see the case $\neg D_m$ in (3)).

– If D_m holds and the method terminates, then the pre-state and its matching post-state satisfy the same condition postcontract(σ, j, m, i) as in the previous case (see the case D_m in (3)).

5 Liveness properties

Liveness properties extend the notion of program termination by stipulating that a program must eventually reach some given states. This section deals with the expression and verification of liveness properties on a class C.

5.1 Liveness operator

The liveness properties under consideration are those expressible by the Loop operator defined in this section. For any state predicate Q, the temporal formula Loop(Q) corresponds to the linear-time temporal logic (LTL) property `GF`$\neg Q$ for infinite sequences of states. It is also satisfied by finite sequences of states ending in a state where Q does not hold. Its semantics is based on the notion of visible states in JML. It is defined on finite and infinite executions as follows:

Definition 12 (loop operator): Let Q be a predicate. The execution $\sigma \in \Sigma_C$ satisfies the liveness operator Loop(Q), written $\sigma \vDash \text{Loop}(Q)$, if

$$\forall i.(0 \le i < \text{len}(\sigma) \Rightarrow \exists j.(i \le j < \text{len}(\sigma) \wedge \sigma(j) \vDash \neg Q)).$$

This satisfaction relation is lifted up to sets of executions with the semantics that every execution in the set satisfies the formula.

5.2 Class liveness

In object-oriented programming, defining and checking the satisfaction of a liveness property on a whole program – composed of many classes – may be an heavy task. As a first step, this section presents the semantics of a liveness property attached to a single Java class.

A liveness property Loop(Q) declared in a class C must hold for every object o of type C. For the sake of simplicity, C is assumed to have no static attribute. Thus, Q is a JML predicate with variables among the (non-static) attributes of C. The satisfaction of Loop(Q) on an execution of Σ_C intuitively means that if, during the execution, any instance of the class C is in a state satisfying Q, then it is always possible to reach a state satisfying $\neg Q$ by invoking methods of C on this instance. In other words, C satisfies the liveness property Loop(Q) if $\Sigma_C \vDash \text{Loop}(Q)$.

5.3 Proving liveness

Along the line of Floyd's total correctness proof method, we plan to prove liveness with the help of a variant function that assigns a value to each program state. That value should decrease at each program step, according to a well-founded ordering. In the deterministic case, it is sufficient [19] to consider variants taking their values in $\mathbb{N}$, totally ordered with $<$.

$$
\begin{array}{ll}
(1) & \sigma : \texttt{invariant}\ I \text{ if } \forall i \ge 0. \sigma(i) \models I. \\
(2) & \begin{array}{l} \sigma : \texttt{constraint}\ H\ \texttt{for}\ M \text{ if} \\ \quad \forall i \ge 0. \forall m \in M.(\mathit{prestate}(\sigma,m,i) \Rightarrow \forall j,k.(\mathit{mp}(\sigma,j,m,i) \wedge i < k \le j \Rightarrow (\sigma(k-1),\sigma(k)) \models H)). \end{array} \\
(3) & \begin{array}{l} \sigma : \texttt{behavior; requires}\ P_m\texttt{; diverges}\ D_m\texttt{; ensures}\ Q_m\texttt{; signals (Exception e)}\ R_m\texttt{;} \text{ if} \\ \ \forall i \ge 0.(\mathit{prestate}(\sigma,m,i) \wedge \sigma(i) \models P_m \Rightarrow \\ \quad (\sigma(i) \models \neg D_m \Rightarrow \exists j > i.(\mathit{mp}(\sigma,j,m,i) \wedge \mathit{postcontract}(\sigma,j,m,i))) \wedge \\ \quad (\sigma(i) \models D_m \Rightarrow \forall j > i.(\mathit{mp}(\sigma,j,m,i) \Rightarrow \mathit{postcontract}(\sigma,j,m,i)))) \end{array}
\end{array}
$$

$$\text{where } \mathit{postcontract}(\sigma,j,m,i) = (\neg\sigma(j)(\mathit{excp}) \Rightarrow (\sigma(i),\ \sigma(j)) \models Q_m) \vee (\sigma(j)(\mathit{excp}) \Rightarrow (\sigma(i),\ \sigma(j)) \models R_m)$$

Figure 3 *Consistency between JML annotations and executions*

In the present case, some program steps are calls to methods of a class C. It is obvious that a call to a side-effect free method of C cannot change the value of any variant. Thus, the variant of a liveness property will be required to decrease strictly for a subset of methods with side effects. Consequently, when assigning a liveness property to a Java class, the user is asked to specify a variant V and a set M of progress methods. This extension of the Loop operator with V and M, attached to a class C, is denoted $\mathrm{Loop}_C(Q, V, M)$.

In order to verify $\Sigma_C \vDash \mathrm{Loop}_C(Q, V, M)$, we need to assume progress of the environment, that is, that the environment invokes the methods of the subset M.

Definition 13 (progress hypothesis): For any set of methods M, an execution $\sigma \in \Sigma_C$ satisfies the progress hypothesis, written $\sigma \vDash PH(M)$, if

$$\forall i.(0 \le i < \mathrm{len}(\sigma) \Rightarrow \exists j.(i \le j < \mathrm{len}(\sigma) \wedge \bigvee_{m \in M} \mathrm{prestate}(\sigma, j, m))).$$

This satisfaction relation extends to sets of executions in a standard way. The semantics of $\mathrm{Loop}_C(Q, V, M)$ is given by the following definition, where 1_M is the characteristic function of set M, whose value $1_M(m)$ at m is 1 if $m \in M$, 0 otherwise.

Definition 14 (liveness clause): Let $C = (V_C, I_C, C_C, M_C)$ be an annotated class, Q a predicate on the attributes of C, $V : V_C \to \mathbb{N}$ a variant function and $M \subseteq M_C$ a set of methods of C. An execution $\sigma \in \Sigma_C$ satisfies the liveness clause $\mathrm{Loop}_C(Q, V, M)$, written $\sigma \vDash \mathrm{Loop}_C(Q, V, M)$, if

$$\sigma \vDash PH(M) \Rightarrow \forall i.((0 \le i < \mathrm{len}(\sigma) - 1) \Rightarrow (\sigma(i) \vDash Q \Rightarrow \bigwedge_{m \in M_C} V(\sigma(i)) - V(\sigma(i+1)) \ge 1_M(m))).$$

The variant-based liveness proof method is summarised in the following proposition.

Proposition 4: For any execution $\sigma \in \Sigma_C$ satisfying the progress hypothesis $PH(M)$, if $\sigma \vDash \mathrm{Loop}_C(Q, V, M)$ then $\sigma \vDash \mathrm{Loop}(Q)$.

5.4 Approximation with JML annotations

This section shows how to use existing JML tools for verifying liveness properties on a class in isolation. The idea is to replace the liveness clause with standard JML annotations, whose satisfaction is sufficient to establish $\Sigma_C \vDash \mathrm{Loop}_C(Q, V, M)$.

Verification of the $\mathrm{Loop}_C(Q, V, M)$ property is quite similar to a termination proof. As long as Q holds, it must be possible to invoke a method of M, and methods in M must decrease the variant V. Here we propose proof obligations – inspired from [20] – expressed as JML annotations. These proof obligations guarantee the satisfaction of the $\mathrm{Loop}_C(Q, V, M)$ property by the executions of the class C in isolation.

Let $\mathcal{A}_{1-5}$ be the following set of JML annotations

$$\texttt{invariant}\ V >= 0; \qquad (\mathcal{A}_1)$$

$$\texttt{constraint}\ Q ==> V < \texttt{\old(}V\texttt{)}\ \ \texttt{for}\ M; \qquad (\mathcal{A}_2)$$

$$\texttt{constraint}\ Q ==> V <= \texttt{\old(}V\texttt{)}; \qquad (\mathcal{A}_3)$$

$$\texttt{invariant}\ Q ==> \bigvee_{m \in M} P_m; \qquad (\mathcal{A}_4)$$

$$\texttt{invariant}\ Q ==> \bigwedge_{m \in M_C} (P_m ==> \texttt{!}D_m); \qquad (\mathcal{A}_5)$$

Remember that JML invariants have to hold on all visible states, and JML constraints have to hold between any two successive visible states [8]. These annotations $\mathcal{A}_{1-5}$ relate to Q, V, M, and a class C and its methods as follows:

$\mathcal{A}_1$ The variant V is actually greater than zero, it is a function returning a natural number.

$\mathcal{A}_2$ As long as Q holds, the variant V must decrease when a method in M is executed.

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

$\mathcal{A}_4$ As long as Q holds, there should always be a method in M that may be called, that is, whose pre-condition P_m (in the clause `requires` P_m) holds. This ensures the deadlock-freeness of the system.

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

In the rest of the paper, $\sigma : \mathcal{A}_{1-5}$ denotes $\sigma : \mathcal{A}_1 \wedge \cdots \wedge \sigma : \mathcal{A}_5$.

Theorem 1: Let $\sigma \in \Sigma_C$ be an execution. If $\sigma : \mathcal{A}_{1-5}$ then $\sigma \vDash \mathrm{Loop}C(Q, V, M)$.

Proof 1: There are two cases:

1. If $\sigma \in \Sigma_C$ is a finite execution, the definitions in Section 4 imply that $\mathrm{last}(\sigma) \nvDash P_m$ for any method m in M_C, and that $\mathrm{last}(\sigma)$ is the prestate of no method. That falsifies $PH(M)$ for any $M \subseteq M_C$ when $i = \mathrm{len}(\sigma) - 1$. Thus $\sigma \vDash \mathrm{Loop}_C(Q, V, M)$.

2. If σ is an infinite execution, the proof is by contradiction. Suppose there exists $\sigma \in \Sigma_C$ such that $\sigma \vDash \mathrm{Loop}_C$

(Q, V, M). By Definitions 13 and 14,

$$\sigma \vDash PH(M) \tag{1}$$

and there are some i, $0 \leq i < \text{len}(\sigma) - 1$ and some method $m \in M$, s.t. $\sigma(i) \vDash Q$ and

$$V(\sigma(i)) - V(\sigma(i+1)) < 1_M(m) \tag{2}$$

Since $\sigma \in \Sigma_C$, by the progress hypothesis (Definition 13), we have

$$\forall k \geq 0.\ \exists k_2 \geq k.\ \exists m \in M.\ \text{prestate}(\sigma, k_2, m)$$

The above property being true for each index $k \geq 0$, it is also the case for each index $k \geq i$

$$\forall k \geq i.\ \exists k_2 \geq k.\ \exists m \in M.\ \text{prestate}(\sigma, k_2, m) \tag{3}$$

Independently, from the semantics of Java statements (Definition 4) and the definition of pre-states (Definition 6), we derive

$$\forall k \geq 0.\ \forall m \in M_C.\ \text{prestate}(\sigma, k, m) \Rightarrow \sigma(k) \vDash P_m \tag{4}$$

On the one hand, from (3) and (4), we obtain

$$\forall k \geq i.\ \exists k_2 \geq k.\ \exists m \in M.\ \text{prestate}(\sigma, k_2, m) \wedge \sigma(k_2) \vDash P_m \tag{5}$$

On the other hand, from (2) and $(\mathcal{A}_5)$, we have

$$\forall m \in M. \sigma(k_2) \vDash P_m \Rightarrow \sigma(k_2) \nvDash D_m \tag{6}$$

Then, from (5) and (6), we obtain

$$\forall k \geq i.\ \exists k_2 \geq k.\ \exists m \in M.\ \text{prestate}(\sigma, k_2, m) \wedge \sigma(k_2) \vDash P_m \wedge \sigma(k_2) \nvDash D_m \tag{7}$$

By Definition 11 (Fig. 3), when using default values [8] of all but D_m of the `behavior` clause on σ above, item (3) results in

$$\forall k \geq i.\ \exists k_2 \geq k.\ \exists m \in M.\ \text{prestate}(\sigma, k_2, m) \wedge \exists k_3 \geq k_2. mp(\sigma, k_3, m, k_2) \tag{8}$$

By $(\mathcal{A}_2)$, (8) and transitivity of history constraints (Definition 11, item (2) in Fig. 3), we obtain

$$\begin{aligned} &\forall k \geq i.\ \exists k_2 \geq k.\ \exists m \in M \\ &\text{prestate}(\sigma, k_2, m) \wedge \exists k_3 \geq k_2.\ mp(\sigma, k_3, m, k_2) \wedge \\ &\langle \sigma(k_2), \sigma(k_3) \rangle \vDash V < \backslash\texttt{old}(V) \end{aligned} \tag{9}$$

By a similar reasoning, we also obtain, from $(\mathcal{A}_3)$

$$\begin{aligned} &\forall k \geq i.\ \exists m \in M_C \\ &\text{prestate}(\sigma, k, m) \Rightarrow \exists j \geq k.\ mp(\sigma, j, m, k) \wedge \\ &\langle \sigma(k), \sigma(j) \rangle \vDash V \leq \backslash\texttt{old}(V) \end{aligned} \tag{10}$$

Consequently, from (9) and (10) one deduces that the variant V decreases infinitely during the execution. And so, $\mathcal{A}_1$ cannot be established. A contradiction. □

In JML side-effect free methods can be identified syntaxically thanks to the keyword `pure`. Let Pure_C be the set of pure methods of the class C. Let $\mathcal{PM}_C = M_C \backslash \text{Pure}_C$ denote the set of the so-called progress methods of the class C, that is, with a side effect. An interesting property is obtained when $M = \mathcal{PM}_C$. In this particular case, the progress hypothesis $PH(M)$ is not only sufficient but also necessary.

Proposition 5: Let $\sigma \in \Sigma_C$ be an execution. If $\sigma \vDash \text{Loop}_C(Q, V, \mathcal{PM}_C)$ and $C : \mathcal{A}_{1-5}$ then $\sigma \vDash PH(\mathcal{PM}_C)$.

6 Liveness temporal patterns

In [7], we have presented a way to verify liveness properties expressed with the Loop_C operator. This section presents a practical context of Java/JML verification where this verification method is applied.

Along the line of helping Java programmers in writing formal specifications, Trentelman and Huisman [4] proposed a temporal extension of JML inspired by the pragmatic work of the SanTos Specification Pattern Project [21]. We refer to this temporal extension of JML as JTPL, for Temporal Pattern Language, prefixed by a 'J' to denote its adaptation to Java. The semantics of temporal formulae in JTPL and translation rules into JML annotations are detailed in [4] for safety properties and in [22] for liveness properties. This section defines a verification technique for liveness properties expressible in JTPL, a problem left open by Trentelman and Huisman [4]. This verification is performed by translating these properties into the Loop_C operator.

6.1 Language overview

JTPL provides the user with patterns to express common temporal requirements of Java classes. Moreover, the language deals with normal and abnormal method terminations. JTPL is based on the notion of trace property which is either always P, eventually P, or the conjunction or disjunction of two trace properties. always P is true on an execution σ if P holds on every state of σ. eventually P is true on an execution σ if P holds on at least one state of σ.

It is often useful to reduce the scope of a trace property, that is, specifying it only for subparts of an execution. This is made possible by the notion of *event*. An event can be: (i) m called, denoting that the method m has been invoked;

(ii) m normal, denoting that the method m has terminated normally, that is, without throwing any exception; (iii) m exceptional, denoting that the method m has terminated by throwing an exception or (iv) m terminates, denoting that the method m has terminated either normally or by throwing an exception.

Now, a temporal property in JTPL is inductively defined as follows: let E be a disjunction of events, C a trace property and T a temporal property. A temporal property can be either: (a) after E T, which is true on an execution σ if the suffix of σ starting after each occurrence of an event in E satisfies the temporal formula T; (b) before E C, which is true on an execution σ if the prefix of σ ending with each occurrence of an event in E satisfies the trace property C; (c) C until E, which is true on an execution σ if an event in E occurs and if the trace property C is satisfied on the segment of σ ending with an event in E; (d) C unless E, which is true on an execution σ if an event in E occurs and the trace property C is satisfied on the segment of σ ending with an event in E, or the trace property C is satisfied on the whole execution σ and E never happens; or (e) between E E' C, which is true on an execution σ if the temporal formula after E (C until E') holds on σ, or (f) a trace property C.

6.1.1 Safety and liveness characterisation:

The properties described by this extension of JML are either safety properties or liveness properties. The following proposition makes it possible to distinguish them syntactically:

Proposition 6 (characterisation of safety and liveness properties): The properties containing only the keywords after, before, unless and always are safety properties. The properties containing the keyword eventually iff they contain the keyword before also are safety properties. The other properties are liveness properties.

For liveness properties, the verification is based on the decrease of a well-founded variant given by the user. Therefore we propose to extend the syntax of liveness formulae with the following clause:

under [invariant <JMLProp>] variant <JMLExpr>
[for <Methods>]

In the above clause, <JMLProp> is a JML predicate which is an optional local invariant – like a loop invariant – that can help the proof, <JMLExpr> is the variant expression (its type is a natural number), and <Methods> is a list of Java method names.

6.1.2 Back to the example:

Using JTPL formulae, one can express the following properties on the `Buffer` example (Fig. 1 Section 2):

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

after `storeData` called always `customized` $\quad (S)$

2. After starting a transaction, that is, after 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`; $\quad (L)$

Property S is a safety property and property L is a liveness property. Notice that in (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 (L) is a liveness property, the user has to give a variant and a set of progress methods with the JTPL clause under variant … for. Here, the variant corresponds to the free space in the `Buffer`, and the for clause contains a list of methods that can potentially modify the value of the variant. So, `storeData` is not in the list.

6.2 Embedding liveness properties into the Loop clause

This section presents a translation of a JTPL liveness property into a Loop$_C$ clause completed with other JML annotations. First, we present the translation for the basic after E eventually P liveness property. Then, we generalise to the other JTPL liveness properties.

Let us consider a temporal formula of the form

after E eventually P under variant V for M $\quad (11)$

To translate liveness JTPL properties, like (11), into a Loop$_C$ clause, one needs to observe whether a particular event has already occurred or whether a state satisfying a predicate has already been reached. For that, we define a `witness` primitive, denoted $\texttt{JML}(X_1, X_2)$, where X_1 and X_2 are either JML predicates or JTPL events. Intuitively, given an execution σ, $\texttt{JML}(X_1, X_2)$ is satisfied on all states of σ between the states satisfying X_1 and X_2.

Definition 15 (witness primitive): Let σ be an execution and i a natural number between 0 and $len(\sigma) - 1$. A state $\sigma(i)$ satisfies $\texttt{JML}(X_1, X_2)$ *iff* $\exists j.(0 \leq j < i \wedge \sigma(j) \vDash X_1 \wedge \forall k.(j < k < i \Rightarrow \sigma(k) \nvDash X_2))$.

The `witness` primitives are expressed by JML `ghost` variables that are assigned w.r.t. events occurring in the formula. The general rules can be easily derived from the following examples:

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. 5) 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`, that is, `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. 5). The `ghost` variable `witness_L` is assigned using a `try` {`try` {T_1} `catch` {T_2} } `finally` {T_3} statement (see annotation L_b). Notice that in the case of exception, the caught exception is re-thrown. The reader can see that `witness_L` is set to `true` only when `begin` normal occurs. The `ghost` variable `witness_L` is set to ¬`trDepth` again by adding a `set` statement (annotation L_c) to each method.

Thanks to an adequate `witness`, one can give a Loop_C clause ensuring property (11). Using the semantics of JTPL in [1] and the semantics in Section 4, one can show that property (11) holds on the execution σ if $\sigma \models \text{Loop}_C$ ($\texttt{JML}(E, P), V, M$).

In a similar way, the other JTPL liveness patterns can be translated into JML annotations (using the Loop_C clause) by the rules given in Fig. 4. For each $\text{Loop}_C(Q, V, M)$, the local invariant J is expressed by an invariant clause `invariant` Q `==>` J. The safety part of the property is also translated into an invariant.

Example 3 (generation of annotations for L): The JML translation of L is

$$\text{Loop}_C(\texttt{witness\_L}, \texttt{getBufferLess}(), \{\texttt{begin}, \texttt{commit}, \texttt{abort}, \texttt{write}\})$$

The corresponding annotations are displayed in Fig. 5 (see annotations L_{loop}). Notice that, since no method of `Buffer` diverges, annotation $\mathcal{A}_5$ does not appear.

7 JML annotation generator

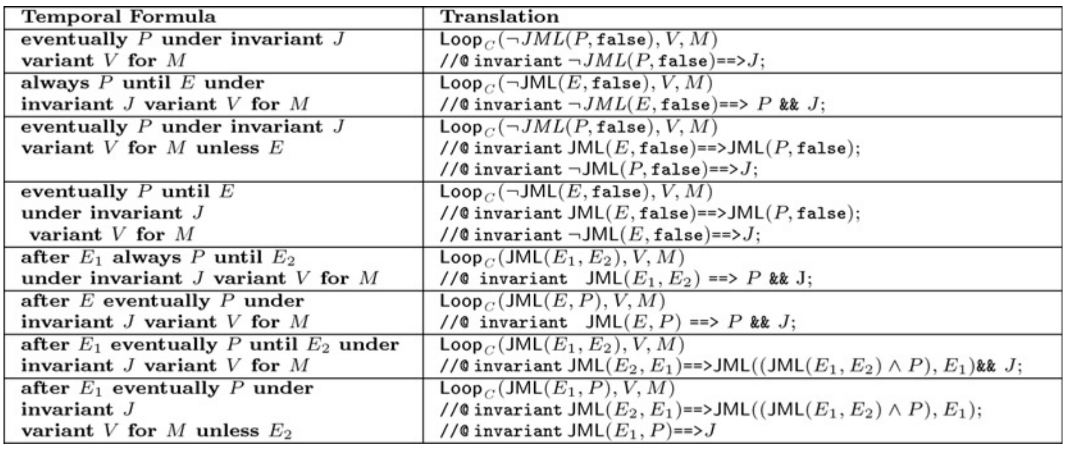
The automatic generation of JML annotations for safety properties in [4] and for liveness properties in Section 5 has been implemented in a tool, called JAG [23]. The JAG release parses a Java file – possibly already JML annotated – with the JML parser included in the Common JML tools and takes a file containing temporal formulae as other input. JAG is freely available from page http://jag.univ-fcomte.fr.

7.1 Translating temporal formulae into intermediate primitives

The tool reduces each temporal property into one or more intermediate primitives, like the `witness` primitive, that are semantically equivalent [4, 22]. These primitives are an internal format which is independent of the JML syntax, allowing an easy extension of the annotation generation to other specification languages, such as Spec♯.

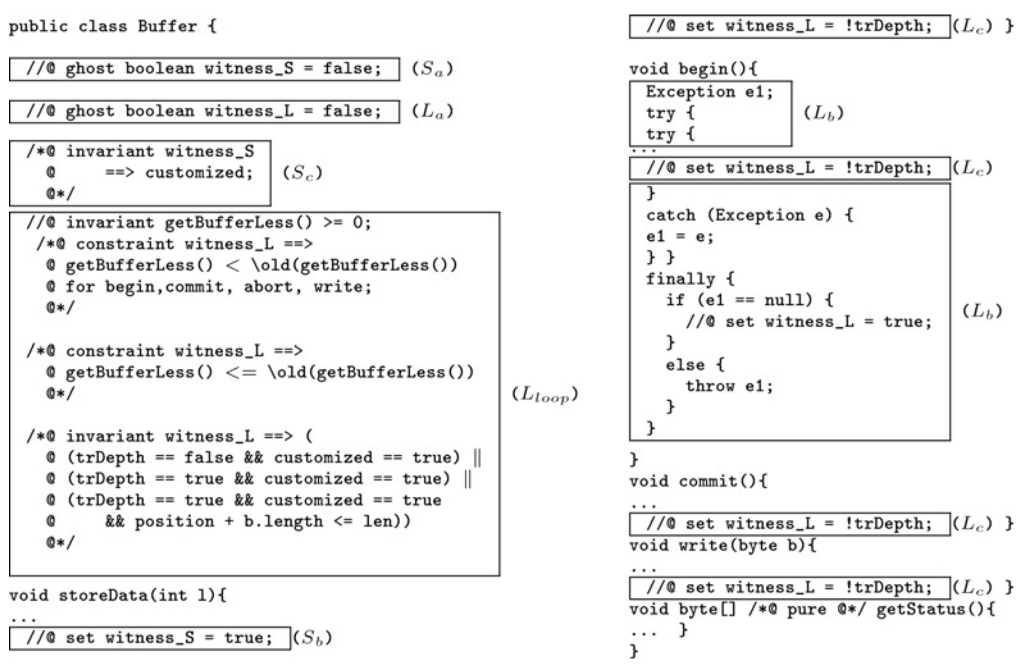
7.2 Translating intermediate primitives into standard JML annotations

Each intermediate `Inv` primitive representing the safety part of a property is translated into a JML `invariant`. Each intermediate Loop primitive representing the liveness part of the property is translated into a set of `invariants` and history `constraints` that imply the decreasing



Temporal Formula	**Translation**
eventually P **under invariant** J **variant** V **for** M	$\text{Loop}_C(\neg JML(P, \texttt{false}), V, M)$ `//@ invariant` $\neg JML(P, \texttt{false})$`==>`$J$;
always P **until** E **under** **invariant** J **variant** V **for** M	$\text{Loop}_C(\neg \text{JML}(E, \texttt{false}), V, M)$ `//@ invariant` $\neg JML(E, \texttt{false})$`==>` P `&&` J;
eventually P **under invariant** J **variant** V **for** M **unless** E	$\text{Loop}_C(\neg JML(P, \texttt{false}), V, M)$ `//@ invariant` $\text{JML}(E, \texttt{false})$`==>`$\text{JML}(P, \texttt{false})$; `//@ invariant` $\neg\text{JML}(P, \texttt{false})$`==>`$J$;
eventually P **until** E **under invariant** J **variant** V **for** M	$\text{Loop}_C(\neg \text{JML}(E, \texttt{false}), V, M)$ `//@ invariant` $\text{JML}(E, \texttt{false})$`==>`$\text{JML}(P, \texttt{false})$; `//@ invariant` $\neg\text{JML}(E, \texttt{false})$`==>`$J$;
after E_1 **always** P **until** E_2 **under invariant** J **variant** V **for** M	$\text{Loop}_C(\text{JML}(E_1, E_2), V, M)$ `//@ invariant` $\text{JML}(E_1, E_2)$ `==>` P `&&` J;
after E **eventually** P **under** **invariant** J **variant** V **for** M	$\text{Loop}_C(\text{JML}(E, P), V, M)$ `//@ invariant` $\text{JML}(E, P)$ `==>` P `&&` J;
after E_1 **eventually** P **until** E_2 **under** **invariant** J **variant** V **for** M	$\text{Loop}_C(\text{JML}(E_1, E_2), V, M)$ `//@ invariant` $\text{JML}(E_2, E_1)$`==>`$\text{JML}((\text{JML}(E_1, E_2) \wedge P), E_1)$`&&` J;
after E_1 **eventually** P **under** **invariant** J **variant** V **for** M **unless** E_2	$\text{Loop}_C(\text{JML}(E_1, P), V, M)$ `//@ invariant` $\text{JML}(E_2, E_1)$`==>`$\text{JML}((\text{JML}(E_1, E_2) \wedge P), E_1)$; `//@ invariant` $\text{JML}(E_1, P)$`==>`J

Figure 4 *Translation of JTPL liveness patterns using the* Loop_C *clause*



```
public class Buffer {

//@ ghost boolean witness_S = false;   (S_a)

//@ ghost boolean witness_L = false;   (L_a)

/*@ invariant witness_S
  @     ==> customized;                (S_c)
  @*/
//@ invariant getBufferLess() >= 0;
 /*@ constraint witness_L ==>
  @ getBufferLess() < \old(getBufferLess())
  @ for begin,commit, abort, write;
  @*/

/*@ constraint witness_L ==>
  @ getBufferLess() <= \old(getBufferLess())
  @*/                                              (L_loop)

/*@ invariant witness_L ==> (
  @ (trDepth == false && customized == true) ||
  @ (trDepth == true && customized == true) ||
  @ (trDepth == true && customized == true
  @     && position + b.length <= len))
  @*/

void storeData(int l){
...
//@ set witness_S = true;   (S_b)
//@ set witness_L = !trDepth;   (L_c) }

void begin(){
Exception e1;
try {                          (L_b)
try {
...
//@ set witness_L = !trDepth;   (L_c)
}
catch (Exception e) {
e1 = e;
} }
finally {
  if (e1 == null) {
    //@ set witness_L = true;    (L_b)
  }
  else {
    throw e1;
  }
}
}
void commit(){
...
//@ set witness_L = !trDepth;   (L_c) }
void write(byte b){
...
//@ set witness_L = !trDepth;   (L_c) }
void byte[] /*@ pure @*/ getStatus(){
...  }
}
```

Figure 5 *Buffer with generated annotations*

of the variant and the deadlock-freeness of the system. Each `witness` is translated into a JML `ghost` variable. Finally, the tool generates an output file including the original file and enriched with the generated JML annotations. Fig. 5 contains the result of the translation of properties S and L.

Example 4 (invariant generation for S): The invariant for S is displayed in Fig. 5 (annotation S_c). It means that when the variable `witness_S` is true, that is, after the first occurrence of `storeData` called, the predicate must be `true` – the definition of property S.

7.3 Trace preservation

The tool is able to keep the trace of the generated annotations, that is, it is possible, given a generated annotation, to find the original intermediate primitive and the original temporal property.

7.4 Experiments

Since the generated output file contains standard JML annotations, it can be used with other JML tools [2] to validate or prove the temporal formulae. For instance, Table 1 – where 'PO' stands for 'Proof Obligation' – summarises the results we have obtained with the JACK tool [12]. All the 277 POs in fourth column have been proved either fully automatically (for 274 POs) or interactively (for remaining three POs by enforcing invariants) with the B4free tool as a back-end theorem prover.

TransactionSystem and AtmTransaction are two academic examples. TransactionSystem is adapted from [4] and inspired by the JavaCard transaction mechanism, that ensures that every transaction in a smart card is atomic. AtmTransaction implements a transactional mechanism between a smard card and a terminal. Notice that our theoretical contributions have been applied not only to that academic examples but also to the Demoney system, a Java Card Electronic Purse application we have developed in the framework of an industrial collaboration with Trusted Logic (http://www.trusted-logic.com/), via the ACI GECCOO project. For this application (whose demonstrative electronic purse – card specification is available at http://www.doc.ic.ac.uk/~siveroni/secsafe/), we wrote over 500 lines of JML annotations.

Table 1 Results for temporal properties verification

Example name	Number of temporal properties to verify	Number of generated annotation lines	Number of POs (automatically proved)
TransactionSystem	2	18	92 (91)
AtmTransaction	2	21	171 (171)
Electronic purse (Demoney)	2	25	14 (12)

Moreover, we have successfully used the JAG tool for the following purposes:

- Verification of the correctness of the Java code w.r.t. the JML annotations with the proof obligation generators Jack [12] and Krakatoa [13];

- Validation of a JML model with JML-TT [24];

- Formal verification of a JML model with the JML2B method [25];

- Test generation and runtime assertion checking with the test generators Tobias [26], Jartege [27] and JML-TT [28].

Test generation and Runtime Assertion Checking using JAG have been studied on an industrial Java Card application [29].

8 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 specify a variant for the verification of a Loop clause to which liveness properties are reduced. The generated JML annotations are verified (or validated) with any tool 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. We are working on extensions of JAG to other temporal properties. In particular, we currently address the verification of properties expressed by Büchi automata. Assuming that a liveness is established on the class in isolation, another challenge is to provide techniques for verifying that the (single- or multi-threaded) environment effectively satisfies a progress hypothesis.

9 Acknowledgments

We would like to thank Pierre Lescanne for his helpful advice, Marieke Huisman for her interesting and helpful comments and suggestions to improve this work, and the anonymous referees for their corrections, comments and advice. This research is partially funded by the French National Research Agency, ANR-06-SETI-017 TACOS.

10 References

[1] BREUNESSE C-B., CATAÑO N., HUISMAN M., JACOBS B.: 'Formal methods for smart cards: an experience report', *Sci. Comput. Program.*, 2005, **55**, (1–3), pp. 53–80

[2] BURDY L., CHEON Y., COK D., *ET AL.*: 'An overview of JML tools and applications'. FMICS 03, 2003, (*LNCS*, **80**), pp. 73–89

[3] LAMPORT L.: 'Proving the correctness of multiprocess programs', *IEEE Trans. Softw. Eng.*, 1977, **3**, (2), pp. 125–143

[4] TRENTELMAN K., HUISMAN M.: 'Extending JML specifications with temporal logic'. AMAST'02, 2002, (*LNCS*, **2422**), pp. 334–348

[5] GIORGETTI A., GROSLAMBERT J.: 'Un programme annoté en vaut deux'. JFLA'07, Journées francophones des langages applicatifs, Aix-les-Bains, France, January 2007 INRIA pp. 87–101

[6] LOGOZZO F.: 'Class invariants as abstract interpretation of trace semantics', *Comput. Lang., Syst. Struct.*, To be published.

[7] GROSLAMBERT J., JULLIAND J., KOUCHNARENKO O.: 'JML-based verification of liveness properties on a class in isolation'. SAVCBS'06, Specification and Verification of Component-Based Systems, Portland, Oregon, USA, November 2006, pp. 41–48

[8] LEAVENS G.T., POLL E., CLIFTON C., *ET AL.*: 'JML Reference Manual'. Department of Computer Science, Iowa State University, Available at http://www.jmlspecs.org, 2003

[9] JACOBS B., MARCHÉ C., RAUCH N.: 'Formal verification of a commercial smart card Applet with multiple tools'. AMAST'04, 2004, (*LNCS*, **3116**), pp. 21–22

[10] MEYER B.: 'Object-Oriented Software Construction' (Prentice Hall, 1997, 2nd rev. edn.)

[11] ROBBY E.R., DWYER M., HATCLIFF J.: 'Checking strong specifications using an extensible software model checking framework'. TACAS 2004, 2004, (*LNCS*, **2988**), pp. 404–420

[12] BURDY L., REQUET A., LANET J.L.: 'Java Applet correctness: a developer-oriented approach'. FM'03, 2003, (*LNCS*, **2805**), pp. 422–439

[13] MARCHÉ C., PAULIN-MOHRING C., URBAIN X.: 'The Krakatoa tool for certification of Java/Java Card programs annotated in JML', *J. Log. Algebr. Program.*, 2004, **58**, (1–2), pp. 89–106

[14] TARSKI A.: 'A lattice-theoretical fixpoint theorem and its applications', *Pac. J. Math.*, 1955, **5**, (2), pp. 285–309

[15] COUSOT P.: 'Constructive design of a hierarchy of semantics of a transition system by abstract interpretation', *Electr. Notes Theor. Comput. Sci.*, 1997, **6**, p. 25

[16] RAGHAVAN A.D., LEAVENS G.T.: 'Desugaring JML method specifications', Technical Report No., 00-03d, Iowa State University, Department of Computer Science, July 2003

[17] VAN DEN BERG J., HUISMAN M., JACOBS B., POLL E.: 'A type-theoretic memory model for verification of

sequential java programs'. WADT, 1999, pp. 1–21, (*LNCS*, **1827**)

[18] LINDHOLM T., YELLIN F.: 'The Java virtual machine specification' 'The Java Series', Addison-Wesley, Reading, MA, USA, 1997

[19] DIJKSTRA E.W.: 'On weak and strong termination', Selected Writings on Computing: A Personal Perspective, 1982, pp. 355–357

[20] BURSTALL R.M.: 'Program proving as hand simulation with a little induction'. IFIP Congress, 1974, pp. 308–312

[21] DWYER M.B., AVRUNIN G.S., CORBETT J.C.: 'Patterns in property specifications for finite-state verification'. In Proceedings of the 21st International Conference on Software Engineering, ICSE '99, Los Angeles, CA, USA, 16–22 May 1999, IEEE Computer Society, Los Alamitos, CA, USA, pp. 411–420

[22] BELLEGARDE F., GROSLAMBERT J., HUISMAN M., JULLIAND J., KOUCHNARENKO O.: 'Verification of liveness properties with JML'. Technical Report RR-5331, INRIA, 2004

[23] GIORGETTI A., GROSLAMBERT J.: 'JAG JML annotation generation for verifying temporal properties'. FASE'2006, Fundamental Approaches to Software Engineering, 2006, (*LNCS*, **3922**), pp. 373–376

[24] BOUQUET F., DADEAU F., LEGEARD B., UTTING M.: 'JML-testing-tools: a symbolic animator for JML specifications using CLP'. TACAS'05 Tool session, 2005, (*LNCS*, **3440**), pp. 551–556

[25] BOUQUET F., DADEAU F., GROSLAMBERT J.: 'Checking JML specifications with B machines'. ZB'05, 2005, (*LNCS*, **3455**), pp. 435–454

[26] LEDRU Y., DU BOUSQUET L., MAURY O., BONTRON P.: 'Filtering TOBIAS combinatorial test suites'. FASE 2004, 2004, (*LNCS*, **2984**), pp. 281–294

[27] ORIAT C.: 'Jartege: a tool for random generation of unit tests for java classes'. SOQUA 2005, 2005, (*LNCS*, **3712**), pp. 242–256

[28] BOUQUET F., DADEAU F., LEGEARD B.: 'Automated boundary test generation from JML specifications'. FM'06, 2006, (*LNCS*, **4085**), pp. 428–443

[29] BOUQUET F., DADEAU F., GROSLAMBERT J., JULLIAND J.: 'Safety property driven test generation from JML specifications'. FATES/RV'06, 2006, (*LNCS*, **4262**), pp. 225–239