View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by INRIA a CCSD electronic archive server

Verification of Liveness Properties with JML

Françoise Bellegarde, Julien Groslambert, Marieke Huisman, Jacques Julliand, Olga Kouchnarenko

To cite this version:

Françoise Bellegarde, Julien Groslambert, Marieke Huisman, Jacques Julliand, Olga Kouchnarenko. Verification of Liveness Properties with JML. [Research Report] RR-5331, INRIA. 2004, pp.24. inria-00071253

HAL Id: inria-00071253

https://hal.inria.fr/inria-00071253

Submitted on 23 May 2006

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

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

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Verification of Liveness Properties with JML

Françoise Bellegarde — Julien Groslambert — Marieke Huisman — Jacques Julliand — Olga Kouchnarenko

N° 5331

October 20, 2004

Thème SYM

ISSN 0249-6399

Verification of Liveness Properties with JML

Françoise Bellegarde, Julien Groslambert, Marieke Huisman, Jacques Julliand, Olga Kouchnarenko

Thème SYM — Systèmes symboliques
Projet Everest, INRIA Sophia Antipolis &
Université de Franche-Comté, LIFC, CNRS, INRIA Projet Cassis

Rapport de recherche n° 5331 — October 20, 2004 — 24 pages

Abstract: This paper proposes a way to verify liveness properties in an extension of JML. The verification is divided into two subtasks: (1) generation of appropriate JML annotations that allow to verify that the class under consideration respects the liveness property, and (2) showing that the environment preserves the liveness properties by proving a refinement. For the generation of appropriate JML annotations, we require that the liveness properties are extended with a *variant* and *invariant* (conform variants and invariants to show termination of loops). We then show that under certain assumptions on the environment, we can prove the satisfaction of the liveness property. The second subtask then boils down to showing that the environment in fact respects these assumptions. The method is illustrated by an example.

Key-words: Verification, liveness, JML

Research partially funded by French Research ACI *Geccoo*.

Vérification des Proprietés de Vivacité avec JML

Résumé : Cet article propose une méthode pour vérifier des propriétés de vivacités dans le cadre d'une extension de JML. Cette vérification est composée de deux sous-tâches: (1) générer les annotations JML appropriées permettant de vérifier que la classe respecte la propriété de vivacité. (2) Montrer que l'environnement préserve la propriété de vivacité par une preuve de raffinement. Pour la génération des annotations JML, la propriété de vivacité doit être complétée avec un variant et un invariant (à l'instar du variant et de l'invariant utilisé dans une preuve de terminaison de boucle). Nous pouvons alors prouver que sous certaines hypothèses sur l'environnement, la propriété de vivacité est satisfaite. La seconde tâche se réduit à prouver que l'environnement respecte les hypothèses énoncées. Nous illustrons notre méthode à l'aide d'un exemple.

Mots-clés : Vérification, vivacité, JML

Contents

1 Introduction

Over recent years, significant progress has been made in the field of program verification, in particular in the domain of (sequential) smart card applications [9, 8, 21]. However, still many obstacles exist that prevent the wide-spread use of these techniques. For example, rigorous development methods such as refinement [1, 3] typically involve the use of mathematical specification languages which are different in kind and more difficult to understand than Java code. Similarly, to describe the temporal behaviour of a system, one needs complicated logics such as LTL (Linear Temporal Logic), CTL (Computational Tree Logic), *etc.* [16]. And to verify that a system adheres to its specification the use of automatic techniques such as model checking [12] is often infeasible, because of the large or infinite number of program states, while interactive proving often requires expertise with a general purpose theorem prover, such as CoQ [6] or B [1].

An interesting development to overcome these problems is the JML project[1]. This defines a specification language which closely resembles Java, thus making specifications more accessible to Java programmers. As part of the project many tools are developed [10]: the JML tool for runtime assertion checking [24]; tools for program verification, *e.g.* Jack [11], ESC/Java2 [13] and Krakatoa [27]; tools for the generation of annotations such as Daikon [17] and Pavlova's generation method [29]. JML allows to specify method's pre- and postconditions, and class invariants, *i.e.,* properties that are preserved by every method in a class. Further, it also provides so-called model variables (specification-only variables), that provide abstraction and make the language expressive enough to specify the order and conditions under which the different methods in a class can be invoked.

However, to specify security policies, one often needs to specify the temporal behaviour of a system, and so-far JML does not support this. In earlier work, Trentelman and Huisman proposed an extension of JML with temporal logic expressions [32, 31] and showed how safety properties in this extension could be translated back into existing JML. However, they do not provide a technique to verify liveness properties, even though these can be expressed in their extension.

This paper addresses this shortcoming by defining a way to verify liveness properties that can be expressed in this JML extension. The basic idea of our approach is to break the verification into two different subtasks. The first subtask is to show that if the class C for which we wish to show the liveness property is run in an ideal environment, *i.e.,* an environment that calls all methods sufficiently often, then the liveness property can be established. The second subtask is to show that the surrounding system, in which the class C is actually used, preserves the liveness property. In this paper, we focus in particular on the first subtask, and we show how to generate JML annotations that are sufficient to guarantee the liveness property, by using appropriate assumptions on the environment. Section 6 discusses the second subtask, which requires to show that the environment satisfies these assumptions, by proving that it is a liveness-preserving refinement of the ideal system, composed with class C. Figure 1 depicts the general approach. Notice that instead of

[1] See `http://www.jmlspecs.org`.

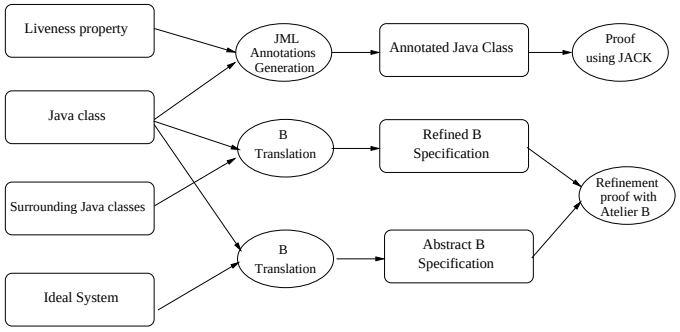

Figure 1: Liveness Verification Methodology

proving the JML annotations with JACK, they also could be validated using any other tool for JML.

To generate appropriate JML annotations, we require that if one specifies a liveness property, two additional kinds of information are given: an invariant and a variant. The invariant describes a property that is maintained until the liveness event actually has happened. The variant must be a well-founded expression that strictly decreases with each method invocation, thus ensuring that "something good eventually will happen". The use of variants and invariants closely resembles the use of loop variants and invariants to show the termination of a while loop. In this paper, we will restrict variants to natural numbers. As a variation, we allow to specify a subset of methods that must decrease the variant; this allows to use pure observer methods in the class under consideration. The ideal system depends on this set of methods. Notice further that the liveness invariant differs from a standard class invariant, in that it only has to be maintained as long as the liveness event has not happened. For this paper, we suppose that methods are atomic, *i.e.*, they do not invoke other methods. However, it is straightforward to generalise our method to allow method invocations of methods on which the liveness property does not depend.

To the best of our knowledge, this is the first time one verifies liveness properties for potentially infinite-state systems using a translation into JML. For finite state systems, liveness properties in PLTL are usually verified automatically by model checkers such as SPIN [18, 19]. For infinite state systems, model checking can be used on a liveness preserving on an abstraction which must be refined later. This supposes a liveness preserving refinement. In our work, we use such a refinement – the B refinement – in order to verify that the environment of the class satisfies the progress assumption. This is similar to the approaches for the verification of PLTL properties of fair systems in TLA [23], UNITY [30] or B [5]. The use of a deductive system for PLTL [26] has the inconvenience of requiring the intervention of experienced users. In this paper, we present an easier verification technique, thanks to the automatic generation of annotations. This is similar to the methodology of

```
//@ model int BufferFree;
//@ initially BufferFree == 10;
//@ ghost boolean TrDepth;
//@ initially !TrDepth;
//@ invariant BufferFree >= 0;
//@ constraint BufferFree >= \old(BufferFree) for commitTransaction;

//@ public normal_behavior    //@ public behavior
//@   requires !TrDepth;      //@   requires BufferFree > 0 && TrDepth;
//@   ensures TrDepth;        //@   ensures BufferFree == 10 && !TrDepth;
void beginTransaction()       //@   signals (Exception) TrDepth;
{//@ set TrDepth = true;...}  void commitTransaction(){...}
```

Figure 2: JML Specification Example

Abrial and Mussat [2], and it can be also compared with the work of Back and Xu [4], who show that verifying liveness preserving simulation requires a well-founded variant.

The rest of this paper is outlined as follows. Sections 2 and 3 present JML and the temporal language extension. Section 4 shows how liveness properties are translated into standard JML annotations, while Section 5 illustrates this on an example. Then, Section 6 discusses the second subtask, *i.e.,* verification of the environment. Finally, Section 7 presents conclusions and future work.

2 A Quick Overview of JML

JML (short for Java Modeling Language) [25, 24] is a specification language especially tailored for Java applications. Originally, JML was proposed by G.T. Leavens and his team; nowadays the development of JML is 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 [9, 8, 21].

JML is developed following the Design by Contract approach [28], 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 specification-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. 2; we refer to [25] for a full overview of the language, and [10] for an overview of different tools using JML.

The keyword `model` allows to declare special JML variables that can be used in specifications only. These variables typically are used as an abstract representation of (some of) the concrete variables (a special `represents` clause allows to state this relation explicitly). Model variables can not be explicitly initialized, instead the `initially` clause gives a predicate constraining their initial values. Notice that the variable declarations further use Java syntax. A variation of model variables are `ghost` variables. These exist also only in spec-

ifications, but a special `set` annotation exists to change their value (while `model` variables change implicitly, when the concrete variables they represent change).

Further we see a declaration of a class `invariant`, denoting a predicate that has to hold before and after every method call, and of a history `constraint`, specifying a relation between the pre- and post-state of a method. A constraint can for example be used to specify that a value only can increase. Constraints and postconditions can use the keyword `\old` to denote the value of an expression, evaluated in the pre-state of a method. The `for` clause specifies to which methods the `invariant` and `constraint` apply.

Finally, the figure contains method specifications for `beginTransaction` and `commitTransaction`. The first is a `normal_behavior` specification: if the method is called in a state satisfying the precondition, it terminates normally in a state satisfying the postcondition. The `behavior` specification means that the method terminates, either in a normal state satisfying the postcondition, or because of an exception, satisfying the exceptional postcondition (`signals` clause).

3 Extending JML with Temporal Specifications

In earlier work, Trentelman and Huisman [32] proposed an extension of JML with temporal logic constructs, inspired by the *Specification Patterns* developed by Dwyer *et al.* [15]. We use this language as a basis for our work, but we extend it by requiring the additional specification of a variant and an invariant for liveness properties, which are used to prove validity of the formula, as described in Sect. 4.

In this section we present (informal) syntax and semantics of this extension, and we discuss how we add the notions of variant and invariant to the specification of the liveness properties.

3.1 Temporal Extension of JML

We present the extension of JML with temporal specifications, and give an informal intuition of their meaning. Below, Sect. 3.4 gives their formal (trace-based) semantics, while Appendix A presents the formal syntax.

In the original extension, special state properties were defined, allowing to specify whether a method is (not) enabled. We do not consider those here, therefore the only *state properties* that we consider are standard JML predicates (not containing `\old`). Let P be a JML predicate. A *trace property* is either:

- **always** P, which is true on an execution σ iff P holds on every state of σ;
- **eventually** P, which is true on an execution σ iff P holds on at least one state of σ; or
- the conjunction or disjunction of two trace properties.

Temporal formulae use sets of events to limit the execution range to which the trace property applies. For each method m, we identify the following events:

- m **called**;
- m **normal**, meaning method m terminates normally;
- m **exceptional**, meaning method m terminates exceptionally; and
- m **terminates**, meaning method m terminates, either normally or exceptionally.

Let E be a set of events and C a trace property. A *temporal property* is either:

- **after** E T, which is true on an execution σ iff occurrence of an event in E implies that the suffix of the execution starting with the event in E satisfies the temporal formula T;
- **before** E C, which is true on an execution σ iff occurrence of an event in E implies that the segment of the executing ending with the event in E satisfies the trace property C;
- C **until** E, which is true on an execution σ iff an event in E occurs and the trace property C is satisfied on the segment of σ ending with an event in E;
- C **unless** E, which is true on an execution σ iff 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 of σ; or
- a trace property C.

3.2 Liveness Properties

The properties described by this extension of JML can be divided into two groups: safety properties, ensuring that nothing bad will happen, and liveness properties, ensuring that something good eventually will happen. For safety properties, a translation into existing JML has been defined [32, 31]; in this paper, we address verification of liveness properties.

We consider as liveness properties all those properties that require to prove that some event eventually will happen. Let P be a JML predicate, C a trace property and E a set of events. Among the temporal properties above, we consider the following as liveness properties.

- **eventually** P;
- **eventually** P **unless** E;
- C **until** E; and
- if T is a liveness property then **after** E T is also a livenes property.

Notice that for a property **eventually** P **until** E, the only liveness-related proof obligation is to show that an event in E occurs; verifying that before this event in E happens, the property P holds is a safety issue.

To be able to verify liveness properties, we require that their specifications are completed by a variant and an invariant. Formally, the syntax for all liveness properties is extended with the following clause.

under invariant <JMLProp> **variant** <JMLVar> [**for** <Methods>]

This annotation induces the following extra proof obligations:

- every method in the class preserves the invariant;
- every method in <Methods> strictly decreases the value of the variant; and
- every method not in <Methods> must not increase the value of the variant.

The set of methods is optional, and its default value is the set of all methods in the class. In general, one would like to specify the set of methods that decrease the variant by excluding any observer methods without side-effects.

To ensure that variant and invariant are given exactly when necessary, the grammar for the JML extension in Appendix A uses attributes [22]. We use two attributes: correct – a synthesized attribute denoting whether the **under** clause is well placed – and inherited – an inherited attribute indicating whether the **until** keyword or the **before** keyword is present in a formula where the **eventually** keyword occurs.

As an example, for the transaction system in Fig. 2, we can specify that `TrDepth` signals whether a transaction is in progress, and that every started transaction eventually will be finished. This can be specified as follows:

```
after beginTransaction called always TrDepth
    until commitTransaction called
under invariant true variant BufferFree
```

To ensure that the system respects this formula, one has to show that for any execution where `beginTransaction` is called, `commitTransaction` will eventually be called, and that as long as `commitTransaction` is not called, `TrDepth` remains true. Section 4 discusses the verification of liveness properties in detail.

3.3 Formal Execution Model

In the next subsection, we give the formal semantics of the JML extension with temporal specifications. This section describes the formal model, based on transition systems, that we use to represent programs. We assume that a translation of Java programs into transition systems exists, see *e.g.* [14] for an example of such a translation.

Definition 1 (Transition System) *Let Ev be a nonempty alphabet of* events *and $\mathcal{P}red$ a set of JML predicates. A labelled and interpreted* transition system *TS is a tuple $< S, S_0, \rightarrow, l >$ composed of a set of states S, a set of initial states $S_0 \subseteq S$, a total labelled transition relation $\rightarrow \subseteq S \times Ev \times S$, and a state labelling function $l : S \rightarrow \mathcal{P}(\mathcal{P}red)$.*

For any JML predicate $P \in \mathcal{P}red$ and any state $s \in S$, we write $s \models P$ whenever $P \in l(s)$. The semantics is defined over executions of transition systems. Executions are infinite sequences of pairs (s, e), where $s \in S$ and $e \in Ev$.

Definition 2 (Execution) *Given a transition system $TS =< S, S_0, \rightarrow, l >$, an* execution *of TS is an infinite sequence $\sigma =_{def} s_0 \xrightarrow{e_1} s_1 \xrightarrow{e_2} s_2 \ldots s_i \xrightarrow{e_{i+1}} s_{i+1} \ldots$ of pairs of states and events such that $s_0 \in S_0$ and for every $i \geq 0$, we have $s_i \xrightarrow{e_{i+1}} s_{i+1}$.*

A JML predicate P holds on execution σ, denoted $\sigma \models P$, if $s_0 \models P$. To define the formal semantics, we also define execution suffixes and segments.

Definition 3 (Execution suffix and Execution segment) *Given an execution σ, the* execution suffix *σ_i denotes the infinite sequence*

$$s_i \xrightarrow{e_{i+1}} s_{i+1} \xrightarrow{e_{i+2}} s_{i+2} \ldots$$

Given an execution σ, the execution segment *σ_i^j denotes the infinite sequence*

$$s_i \xrightarrow{e_{i+1}} s_{i+1} \ldots s_j \xrightarrow{e_j} s_j \xrightarrow{e_j} s_j \ldots$$

Naturally, P holds on σ_i and σ_i^j if $s_i \models P$. Finally, we formally define what we mean be occurrence of an event in E.

Definition 4 (Occurrence of events) *Let $E \subseteq Ev$ be a set of events and σ an execution. We say an event in E* occurs *in state s_i, written $s_i \models E$, if $e_i \in E$.*

We say an event e occurs *on σ if there exists an i such that $s_i \models e$. We say an event e does* not occur *on σ if for all i we have $s_i \not\models e$.*

3.4 Formal Semantics

Now we can give a formal semantics of trace properties and temporal properties.

Definition 5 (Semantics of trace properties) *Let σ be an execution and P a state property. The satisfaction relation of trace properties is inductively defined as follows*

$$\begin{array}{lll} \sigma \models \textbf{always } P & \text{iff} & \forall j.\, j \geq 0 \Rightarrow \sigma_j \models P \\ \sigma \models \textbf{eventually } P \textbf{ [under invariant } J & \text{iff} & \exists j.\, j \geq 0 \wedge \sigma_j \models P \\ \quad \textbf{variant } V \textbf{ for } M] & & \end{array}$$

Definition 6 (Semantics of temporal formulae) *Let σ be an execution, E a set of events, T a temporal formula, and C a trace property. The satisfaction relation for temporal formulae is inductively defined as follows.*

$$\begin{array}{lll}
\sigma \models \textbf{after } E\ T & \text{iff} & \forall j.\ j \geq 0 \Rightarrow \sigma_j \models E \Rightarrow \sigma_j \models T \\
\sigma \models \textbf{before } E\ C & \text{iff} & \forall j.\ j \geq 0 \Rightarrow \sigma_j \models E \Rightarrow \sigma_0^{j-1} \models C \\
\sigma \models C \textbf{ until } E \textbf{ under invariant } J & \text{iff} & \exists j.\ j \geq 0 \wedge \sigma_j \models E \wedge \sigma_0^{j-1} \models C \\
\quad \textbf{variant } V\ [\textbf{for } M] & & \\
\sigma \models C \textbf{ unless } E & \text{iff} & (\sigma \models C \wedge \sigma \not\models E) \vee \\
 & & (\exists j.\ j \geq 0 \wedge \sigma_j \models E \wedge \sigma_0^{j-1} \models C)
\end{array}$$

Notice that J, V, and M are not present in the semantics of **until** and **eventually**; we only use them to generate the appropriate proof obligations to verify the liveness properties.

4 Generation of JML Annotations for Liveness Properties

As explained above, to verify that a liveness property specified in a class C is satisfied, we generate appropriate JML annotations for C, which can be checked with any existing tool for JML. Further, we make assumptions on the environment of class C, which are sufficient to ensure that if C satisfies the generated annotations, then C run in its environment satisfies the liveness property. This section discusses the assumptions on the environment and the JML annotations; Sect. 6 discusses how the assumptions on the environment can be verified.

Remember that for the time being, we assume all methods in C to be atomic, *i.e.,* they do not call any other method.

4.1 Hypotheses on the Environment

For the generation of appropriate JML annotations for class C, the following hypotheses are made about the behaviour of the environment of C.

- Executions are sequences of method calls belonging to C.
- All exceptions thrown by methods in C are caught by the environment.
- The environment is fair, *i.e.,* if there is a set of methods that can be invoked (because their precondition holds) one of these methods effectively is invoked.
- Executions are infinite.
- If a liveness property is restricted by a clause **for** M, then the environment invokes infinitely often the methods in M.

4.2 Proof Obligations for the Primitive *Until* Operator

To generate JML annotations for liveness properties, we first introduce a primitive operator *Until*, and we show how to generate appropriate JML annotations for this primitive. Next, we show how liveness property can be expressed in terms of this operator, leading immediately to appropriate JML annotations.

The primitive operator *Until*, denoted $Q \rightsquigarrow_{JML} P(J, V, M)$, where P, Q and J are JML predicates, V a JML expression returning an integer, and M a set of methods, intuitively holds on σ, if for all suffixes σ_i such that Q holds in s_i, we can find a later state where P holds, and all intermediate states satisfy Q. Formally we define this as follows.

Definition 7 (*Until*) *Let P, Q and J be JML predicates, M a set of methods, and V a JML expression returning an integer. Then $Q \rightsquigarrow_{JML} P(J, V, M)$ holds on excecution σ if*

$$\forall i.i \geq 0 \wedge \sigma_i \models Q \wedge J \;\Rightarrow\; \exists j.j > i \wedge \sigma_j \models P \wedge\; \forall k.i < k < j \Rightarrow \sigma_k \models Q \wedge J$$

Notice that implicitly σ is supposed to satisfy the hypotheses described in Sect. 4.1, thus calling the methods in M infinitely often. Notice also that the invariant J is required everywhere Q holds, up to the moment P holds.

Following [2, 1], we can find proof obligations that are sufficient to guarantee satisfaction of this operator. First we express these proof obligations in terms of executions, then we show the corresponding JML annotations.

Proposition 1 (Proof obligations *Until*) *Let P, Q and J be JML predicates, V a JML expression of type integer, and M a method set. Then $Q \rightsquigarrow_{JML} P(J, V, M)$ holds on execution σ if the following proof obligations are satisfied*[2]*.*

$$\forall i.\, \sigma_i \models Q \Rightarrow \sigma_i \models J \tag{1}$$

$$\forall i.\, \sigma_i \models V \in \mathbb{N} \tag{2}$$

$$\forall i\, n.\, \sigma_i \models Q \wedge V = n \Rightarrow \sigma_{i+1} \models \neg P \wedge \mathsf{term}(M) \Rightarrow \sigma_{i+1} \models V < n \tag{3}$$

$$\forall i\, n.\, \sigma_i \models Q \wedge \sigma_i \models V = n \Rightarrow \sigma_{i+1} \models \neg P \wedge \mathsf{term}(\overline{M}) \Rightarrow \sigma_{i+1} \models V \leq n \tag{4}$$

$$\forall i.\, \sigma_i \models Q \wedge J \Rightarrow \sigma_{i+1} \models \neg P \;\Rightarrow \exists m.\, \sigma_{i+1} \models \mathsf{requires}(m) \wedge \neg \mathsf{diverges}(m) \tag{5}$$

Notice that these proof obligations basically describe a termination proof, using invariant J and variant V, from which we can conclude that eventually P should hold. Intuitively, these proof obligations should be understood as follows.

1. The invariant J has to hold, whenever Q holds.
2. The variant V actually expresses a natural number, *i.e.*, it is well-founded.

[2] Where $\mathsf{term}(M)$ is the set of events $\{m$ **terminates**$, m$ **exceptional**$, m$ **normal** $\mid m \in M\}$, $\mathsf{requires}(m)$ denotes the precondition, and $\mathsf{diverges}(m)$ the diverges clause –the conditions under which the method might diverge –, of method m, respectively.

3. If Q holds, and a method in M is called, the variant V must strictly decrease. This ensures that progress will be made. Notice that because we work with atomic methods, the variant V only has to decrease after termination events.
4. If Q holds, and a method not in M is called, the variant V may not increase.
5. As long as Q holds, and P is not reached, there always should be a method enabled (*i.e.,* its precondition holds, and it will not diverge). This ensures the system will not block.

Following Proposition 1 and the JML semantics [20], we can find JML annotations that are sufficient to imply validity of the *Until* primitive.

Proposition 2 (JML Proof Obligations *Until*) *Validity of the following JML assertions implies that* $Q \leadsto_{JML} P(J,V,M)$ *holds.*

1. `//@ invariant` Q `==>` J
2. `//@ invariant` V `>= 0`
3. `//@ constraint \old(`Q`) & \old(`J`) & !`P `==>` V `< \old(`V`) for` M
4. `//@ constraint \old(`Q`) & \old(`J`) & !`P `==>` V `<= \old(`V`) for` $\overline{M}$
5. `//@ constraint \old(`Q`) & \old(`J`) & !`P `==>` $\exists m.\, \mathsf{requires}(m)$ `&` $\neg\mathsf{diverges}(m)$

4.3 JML Annotations for "after E_1 always JP until E_2"

As explained above, we express each liveness property in the JML extension in terms of the *Until* primitive, which immediately gives us the appropriate JML annotations. This section discusses the instantiation of the property **after** E_1 **always** JP **until** E_2 **under invariant** J **variant** V **for** M. We use $\mathsf{JML}(E)$ to denote the ghost variables associated to the events in the set E (cf. [32]).

Proposition 3 *Let* JP *be a JML property and* E_1, E_2 *be sets of events. The temporal property* **after** E_1 **always** JP **until** E_2 **under invariant** J **variant** V **for** M *holds on execution* σ *if* $\mathsf{JML}(E_1) \leadsto_{JML} \mathsf{JML}(E_2)(J \wedge JP, V, M)$ *holds on* σ.

Proof.
By Definition 7, $\mathsf{JML}(E_1) \leadsto_{JML} \mathsf{JML}(E_2)(J \wedge JP, V, M)$ holds on σ if

$$\begin{array}{ll} \forall j. & j \geq 0 \Rightarrow \sigma_j \models E_1 \wedge J \wedge JP \Rightarrow \\ & \quad \exists k.(k > j \wedge \sigma_k \models E_2 \ \wedge \forall i.\, j \leq i < k \Rightarrow \sigma_i \models E_1 \wedge J \wedge JP) \end{array}$$

Notice that

$$\forall i.\, j \leq i < k \Rightarrow \sigma_i \models JP \Rightarrow \ \forall i.\, j \leq i \Rightarrow \sigma_i^{k-1} \models JP$$

Temporal Formula	**Proof Obligation**
eventually P **under invariant** J **variant** V **for** M	$\neg$`Pa` $\leadsto_{JML} P(J,V,M)$
always P **until** E **under invariant** J **variant** V **for** M	$\neg$JML$(E) \leadsto_{JML}$ JML$(E)(J \wedge P, V, M)$
eventually P **under invariant** J **variant** V **for** M **unless** E	$\neg$`Pa` $\leadsto_{JML} P(J,V,M)$ `//@ constraint` JML(E)`==>\old(Pa);`
eventually P **until** E **under invariant** J **variant** V **for** M	$\neg$JML$(E) \leadsto_{JML}$ JML$(E)(J,V,M)$ `//@ constraint` JML(E)`==>\old(Pa);`
after E_1 **always** P **until** E_2 **under invariant** J **variant** V **for** M	JML$(E_1) \leadsto_{JML}$ JML$(E_2)(J \wedge P, V, M)$
after E **eventually** P **under invariant** J **variant** V **for** M	$\neg$`Pe` $\wedge$ JML$(E) \leadsto_{JML} P(J,V,M)$
after E_1 **eventually** P **until** E_2 **under invariant** J **variant** V **for** M	JML$(E_1) \wedge \neg$`Pe` $\leadsto_{JML}$ JML$(E_2)(J,V,M)$ `//@ constraint` JML(E_2)`==>\old(Pe);`
after E_1 **eventually** P **under invariant** J **variant** V **for** M **unless** E_2	JML$(E_1) \leadsto_{JML} P(J,V,M)$ `//@ constraint` JML(E_2)`==>\old(Pe);`

Figure 3: Translation of Liveness Properties into *Until* Primitive

because the hypothesis of this formula implies that every state s_i for $i \in [j,k)$ satisfies JP, and thus any segment trace σ_i^{k-1} for $i > j$ only contains states satisfying JP. Using this, we can conclude

$$\forall j.\, j \geq 0 \Rightarrow \sigma_j \models E_1 \Rightarrow \exists k.(k > j \wedge \sigma_k \models E_2 \wedge \forall i.\, j < i \Rightarrow \sigma_i^{k-1} \models JP$$

which is the semantics of **after** E_1 **always** JP **until** E_2 **for** M. □

4.4 Generation of JML Annotations for Liveness Properties

In a similar way, we translate all other liveness properties into the *Until* primitive (see Fig. 3). For this we use two auxiliary ghost variables, `Pa` and `Pe`, with the following behavior: `Pa` becomes `true` when the current state satisfies P and then remains `true`; `Pe` becomes `true` when the current state satisfies P and a particular event E has happened, and then remains `true`. The behaviour of `Pa` and `Pe` is formally described by the following JML annotations, where the (exceptional) postcondition is added to all methods in the class.

JML Annotations `Pa`	**JML Annotations for** `Pe`
`//@ ghost boolean Pa;` `//@ initially P == Pa;`	`//@ ghost boolean Pe;` `//@ initially !Pe;`
`//@ ensures P ==> Pa;`	`//@ ensures P & JML(E) ==> Pe;`
`//@ signals(Exception)P ==> Pa;`	`//@ signals(Exception)P & JML(E) ==> Pe;`
`//@ ensures \old(Pa) ==> Pa;`	`//@ ensures \old(Pe) ==> Pe;`
`//@ signals(Exception)\old(Pa) ==> Pa;`	`//@ signals(Exception)\old(Pe) ==> Pe;`

5 Example

To illustrate our method, we discuss the verification of a liveness property on a class implementing a transaction system (loosely inspired by the transaction mechanism of Java Card). When introducing JML in Sect. 2, Fig. 2 already showed a fragment of the example; Fig. 4 specifies all relevant methods. The boxed annotations are generated from the liveness property, as described below.

The transaction mechanism that we specify works as follows: a call to the method `beginTransaction` initiates a set of updates. Each update is performed via the method `modify`. Updates are stored in a buffer, whose state can be observed by the method `getBufferFree`. Eventually, all modifications can be committed by calling the method `commitTransaction`. In case the buffer is full, the method `abortTransaction` is called to abort the transaction.

To model the buffer, we use the integer model variable `BufferFree` to denote the free space in the buffer. We wish to verify that after `beginTransaction` is called, the variable `TrDepth` remains true until `abortTransaction` or `commitTransaction` is called. Notice that implicitly this implies a liveness property: after invoking the method `beginTransaction`, either `commitTransaction` or `abortTransaction` has to be called eventually. This property can be expressed in JML's temporal logic extension as follows.

```
after beginTransaction called always TrDepth
  until commitTransaction called, abortTransaction called
under invariant true variant BufferFree
for beginTransaction, commitTransaction, abortTransaction, modify;
```
(6)

Notice that `getBufferFree` is not involved in the liveness property, thus we assume that there is no infinite sequence of `getBufferFree` invocations, and we do not require that `getBufferFree` decreases the variant.

5.1 Generation of Annotations

Using the *Until* modality described in Sect. 4, we generate JML annotations for this temporal formula. The temporal formula is of the form:

$$\textbf{after } E_1 \textbf{ always } JP \textbf{ until } E_2 \textbf{ under invariant } J \textbf{ variant } V \textbf{ for } M$$

Thus, following Sect. 4, we have to verify the following modality:

$$\texttt{BTcalled} \rightsquigarrow_{JML} (\texttt{ATcalled} \mid\mid \texttt{CTcalled})(\texttt{true} \wedge \texttt{TrDepth}, \texttt{BufferFree}, M)$$

where $M = \{\texttt{beginTransaction}, \texttt{commitTransaction}, \texttt{abortTransaction}, \texttt{modify}\}$, and `BTcalled`, `CTcalled` and `ATcalled` are ghost variables generated to capture the methods' event behaviours (cf. [32]). Following Proposition 2, this gives rise to the JML assertions displayed in Fig. 4. The consistency of this specification can be verified using any tool for JML; we discharged all proof obligations automatically using JACK [11].

```
//@ model int BufferFree;
//@ initially BufferFree == 10;
//@ model boolean TrDepth;
//@ initially !TrDepth;
 //@ ghost bool BTcalled = false;
 //@ ghost bool ATcalled = false;
 //@ ghost bool CTcalled = false;
//@ invariant BTcalled ==> true; // Proof Obligation 1
//@ invariant BufferFree >= 0; // Proof Obligation 2
// Proof Obligation 3
 //@ constraint \old(BTcalled) & \old(true) & !ATcalled & !CTcalled)
     ==> variant(BufferFree) < variant(\old(BufferFree))
     for beginTransaction, commitTransaction, modify, abortTransaction;
// Proof Obligation 4
 //@ constraint \old(BTcalled) & \old(true) & !ATcalled & !CTcalled)
     ==> variant(BufferFree) <= variant(\old(BufferFree)) for getBufferFree;
// Proof Obligation 5
 //@ constraint (\old(BTcalled) & \old(true) & !ATcalled & !CTcalled)
     ==> !TrDepth | (BufferFree > 0 & TrDepth) | (BufferFree == 0 & TrDepth)
//@ public normal_behavior
//@   requires !TrDepth;
//@   ensures TrDepth;
void beginTransaction()
{ //@ set BTcalled = true; set CTcalled = false; set ATcalled = false; ...};

//@ public normal_behavior
//@   requires BufferFree > 0 & TrDepth;
//@   ensures BufferFree == 10 & !TrDepth;
void commitTransaction()
{ //@ set BTcalled = false; set CTcalled = true; ...};

//@ public normal_behavior
//@   requires BufferFree == 0 & TrDepth;
//@   ensures BufferFree == 10 & !TrDepth;
void abortTransaction()
{ //@ set BTcalled = false; set ATcalled = true; ...};

//@ public normal_behavior
//@   requires true;
//@   ensures \result == BufferFree;
int getBufferFree(){...};

//@ public normal_behavior
//@   requires BufferFree > 0 & TrDepth;
//@   ensures BufferFree == \old(BufferFree) - 1;
void modify(){...};
```

Figure 4: Transaction System Example

Model	# proof obligations	# interactive proofs
Abstract	18	6 (5 min. to construct proofs)
Refined	78	28 (10 min. to construct proofs)

Table 1: Statistics Concerning Refinement Proof

6 Verification of the Environment

As explained in Sect. 1 above, our method to verify liveness properties consists of two subtasks. The first task requires to show that the class itself establishes the liveness property by verifying appropriate JML annotations, as discussed in Sect. 4; the second task requires to show that the class is run in an appropriate environment, *i.e.*, an environment that will actually enable it to establish the liveness property, by calling the appropriate methods often enough. Suppose for example that the code in Fig. 4 is run in the following environment.

```
public void main(String [] args){beginTransaction(); modify();}
```

No execution of this system satisfies liveness property (6), even though the implementation of the class respects the invariants and constraints specified. To disallow this kind of environment, we require that any environment in which the class is used is a refinement of the *ideal* environment, *i.e.*, the environment that invokes all relevant methods infinitely often. We propose to use the B refinement [1] for this verification, as this is a liveness-preserving refinement. More precisely, we translate the concrete system, consisting of the class and the concrete environment, and the ideal system, only consisting of the class [3], into B-models, and we show that the concrete system is a refinement of the ideal system. Correctness of the refinement implies preservation of the temporal property in the refined system. As an example, we used this method to prove that temporal formula (6) is satisfied when the class is run in the following environment:

```
public void main(String [] args){
  int i = 16; int m = 0;
  beginTransaction();
  while (m <= i && m < 10) modify();
  if (m == 10) {abortTransaction();} else commitTransaction();}
```

We translated the JML class and this environment into B models (given in Appendix B). Table 1 summarises statistics concerning the refinement proof. Notice that all interactive proofs were straightforward.

7 Conclusion and Future Work

This paper presents a way to verify liveness properties specified in JML's temporal logic extension [32], by generating appropriate JML annotations. This requires that the user

[3] Following the B method, the ideal environment does not need to be specified explicitly.

specifies a variant and invariant. The generated JML annotations can be verified (or validated) with any tool handling JML. Additionally, the verification method requires to show that the environment in which a class is used respects appropriate *progress* hypotheses.

In the near future we plan to develop a tool that generates JML annotations for the liveness properties, and to use this tool on several case studies, to show the validity of our approach.

For the verification that the environment respects the appropriate progress conditions, we have sketched an approach in Sect. 6. However, it is a future work to define a systematic method that allows to show that a liveness property is preserved on infinite traces involving other methods than those directly related to the property. The challenge is to verify the appropriate integration of the classes in the environment.

Section 6 suggests an answer based on the the following observation. If we have a class C, used in an environment S, then the infinite traces of $S + C$ are a special kind of refinement of the infinite traces involving only methods in C. Therefore, we can verify whether the infinite traces of $S + C$ refine the infinite traces of C for a non-divergent τ-simulation, a property of the B event system refinement (see [7]). This would make it possible to use the B prover, provided that we have a sound translation of $S + C$ into a B event system.

Another possibility, which seems easier at first, could be to check the progress hypotheses H directly during the integration of C into S. This would require to express H in a temporal logic which can be either effectively checked on $S + C$ or model checked on a suitable finite abstraction of $S + C$.

References

[1] J.-R. Abrial. *The B Book.* Cambridge University Press, 1996.

[2] J.-R. Abrial and L. Mussat. Introducing dynamic constraints in B. In D. Bert, editor, *B'98 : The 2nd International B Conference*, number 1393 in LNCS, pages 83–128. Springer, 1998.

[3] R.-J. Back and J. von Wright. *Refinement Calculus: A Systematic Introduction.* Graduate Texts in Computer Science. Springer, 1998.

[4] R.-J. Back and Q. Xu. Refinement of fair action systems. *Acta Informatica*, 35:131–165, 1998.

[5] H.R. Barradas and D. Bert. Specification and proof of liveness properties under fairness assumption in B event systems. In *IFM'02*, number 1993 in LNCS. Springer, 2002.

[6] B. Barras, S. Boutin, C. Cornes, J. Courant, J.-C. Filliâtre, H. Herbelin, G. Huet, P. Manoury, C. Muñoz, C. Murthy, C. Parent, C. Paulin-Mohring, A. Saïbi, and B. Werner. *The Coq Proof Assistant Reference Manual Version 6.1.* INRIA-Rocquencourt-CNRS-ENS Lyon, December 1996.

[7] F. Bellegarde, J. Julliand, and O. Kouchnarenko. Ready-simulation is not ready to express a modular refinement relation. In *Fundamental Aspects of Software Engineering, FASE'2000*, number 1783 in LNCS, pages 266–283. Springer, 2000.

[8] L. du Bousquet, Y. Ledru, O. Maury, C. Oriat, and J.-L. Lanet. A case study in JML-based software validation. In *Automated Software Engineering*, 2004.

[9] C. Breunesse, N. Cataño, M. Huisman, and B. Jacobs. Formal methods for smart cards: an experience report. Technical Report NIII-R0316, NIII, University of Nijmegen, 2003. To appear in *Science of Computer Programming*.

[10] L. Burdy, Y. Cheon, D. Cok, M. Ernst, J. Kiniry, G.T. Leavens, K.R.M. Leino, and E. Poll. An overview of JML tools and applications. In Th. Arts and W. Fokkink, editors, *Eighth International Workshop on Formal Methods for Industrial Critical Systems (FMICS 03)*, volume 80 of *ENTCS*, pages 73–89. Elsevier, 2003.

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

[12] E.M. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 1999.

[13] D. Cok and J. Kiniry. ESC/Java2: Uniting ESC/Java and JML. In *Construction and Analysis of Safe, Secure and Interoperable Smart devices (CASSIS'04)*, LNCS. Springer, 2004. To appear.

[14] J. Corbett, M.B. Dwyer, J. Hatcliff, S. Laubach, C. Păsăreanu, Robby, and H. Zheng. Bandera: Extracting finite-states models from Java source code. In *22nd International Conference on Software Engineering*, pages 439–448, 2000.

[15] M.B. Dwyer, G.S. Avrunin, and J.C. Corbett. Patterns in property specifications for finite-state verification. In *International Conf. on Software Engineering*, pages 411–420. IEEE Computer Society Press/ACM Press, 1999.

[16] E.A. Emerson. Temporal and modal logic. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B: Formal Models and Semantics, chapter 14, pages 996–1072. Elsevier, 1990.

[17] M.D. Ernst, J. Cockrell, W.G. Griswold, and D. Notkin. Dynamically discovering likely program invariants to support program evolution. *IEEE Transactions on Software Engineering*, 27(2):1–25, 2001.

[18] R. Gerth, D. Peled, M. Vardi, and P. Wolper. Simple on-the-fly automatic verification of linear temporal logic. In P. Dembinski and M. Sredniawa, editors, *Protocol Specification, Testing and Verification (PSTV'95)*, 1995.

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

[20] M. Huisman. *Reasoning about JAVA programs in higher order logic with PVS and Isabelle.* PhD thesis, University of Nijmegen, 2001.

[21] B. Jacobs, C. Marché, and N. Rauch. Formal verification of a commercial smart card applet with multiple tools. In *Algebraic Methodology and Software Technology (AMAST'04)*, number 3116 in LNCS, pages 21–22. Springer, 2004.

[22] D.-E. Knuth. *The Art of Computer Programming vol. 3.* Addison Wesley, 1973.

[23] L. Lamport. The temporal logic of actions. *ACM Transactions on Programming Languages and Systems*, 16(3):872–923, May 1994.

[24] G.T. Leavens, A.L. Baker, and C. Ruby. Preliminary design of JML: a behavioral interface specification language for Java. Technical report, Iowa State University, Dept. of Computer Science, 1998 revision 2001.

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

[26] Z. Manna and A. Pnueli. How to cook a proof system for your pet language. In *ACM Symp on Principle of Programming Languages*, pages 141–154, 1983.

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

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

[29] M. Pavlova, G. Barthe, L. Burdy, M. Huisman, and J.-L. Lanet. Enforcing high level security properties for applets. In J.-J. Quisquater, P. Paradinas, Y. Deswarte, and A.A. El Kalam, editors, *Cardis'04*, pages 1–16. Kluwer, 2004.

[30] B.A. Sanders. On the UNITY design decisions. In *Research Directions in High-Level Parallel Programming Languages*, number 574 in LNCS, pages 50–63. Springer, 1991.

[31] K. Trentelman. *Aspects of Java Specification and Verification.* PhD thesis, RSISE, Australian National University, 2004. Manuscript.

[32] K. Trentelman and M. Huisman. Extending JML Specifications with Temporal Logic. In H. Kirchner and C. Ringeissen, editors, *Algebraic Methodology And Software Technology (AMAST'02)*, number 2422 in LNCS, pages 334–348. Springer, 2002.

A Syntax of Temporal Extension JML

A.1 Attribute Grammar

(TF1)	<Temp$_1$>	::=	**after** <Events> <Temp$_2$>
(TF2)		|	**before** <Events> <TraceProp>
(TF3)		|	<TraceProp> **until** <Events> **under invariant** <JMLProp> **variant** <JMLvar> [**for** <Methods>]
(TF4)		|	<TraceProp> **unless** <Events>
(TF5)		|	<TraceProp>
(TP1)	<TraceProp$_1$>	::=	**always** <JMLProp>
(TP2)		|	**eventually** <JMLProp> **under invariant** <JMLProp> **variant** <JMLvar> [**for** <Methods>]
(TP3)		|	**eventually** <JMLProp>
(TP4)		|	<TraceProp$_2$> **&** <TraceProp$_3$>
(TP5)		|	<TraceProp$_2$> | <TraceProp$_3$>

<Events>	::=	<Event>		<Methods>	::=	<Method>
	|	<Event>,<Events>			|	<Method>,<Methods>
<Event>	::=	<method> **called**				
	|	<method> **normal**				
	|	<method> **exceptional**				
	|	<method> **terminates**				

A.2 Attribute Definitions.

(TF1)	correct(<Temp$_1$>)	$\leftarrow$	correct(<Temp$_2$>)
(TF2,3,4,5)	correct(<Temp$_1$>)	$\leftarrow$	correct(<TraceProp>)
(TF2,3)	inherited(<TraceProp>)	$\leftarrow$	true
(TF4,5)	inherited(<TraceProp>)	$\leftarrow$	false
(TP1)	correct(<TraceProp$_1$>)	$\leftarrow$	true
(TP2)	correct(<TraceProp$_1$>)	$\leftarrow$	$\neg$inherited(<TraceProp$_1$>)
(TP3)	correct(<TraceProp$_1$>)	$\leftarrow$	inherited(<TraceProp$_1$>)
(TP4,5)	correct(<TraceProp$_1$>)	$\leftarrow$	correct(<TraceProp$_2$>) $\wedge$ correct(<TraceProp$_3$>)
(TP4,5)	inherited(<TraceProp$_2$>)	$\leftarrow$	inherited(<TraceProp$_1$>)
(TP4,5)	inherited(<TraceProp$_3$>)	$\leftarrow$	inherited(<TraceProp$_1$>)

B B models

We show the B code modeling the transaction system and its integration, described in Sect. 6.

B.1 Abstract System

```
MACHINE
   Transaction
VARIABLES
   BufferFree,TrDepth
INVARIANT
   BufferFree : NAT & TrDepth : BOOL
INITIALISATION
BufferFree := 10 || TrDepth := FALSE
OPERATIONS
BT = SELECT TrDepth = FALSE THEN TrDepth := TRUE END;

CT = SELECT (TrDepth = TRUE & BufferFree > 0)
     THEN TrDepth := FALSE || BufferFree := 10 END;

AT = SELECT (TrDepth = TRUE & BufferFree = 0 )
     THEN TrDepth := FALSE || BufferFree := 10 END;

Modify = SELECT (TrDepth = TRUE & BufferFree > 0)
                 THEN BufferFree := BufferFree - 1 END;
END
```

B.2 Refined System

To simulate the execution of the system, we have to model the method invocations made by the environment. The main idea is to see this class as a B event system. We split the Java environment into several new B operations – in this case: `main1`, executing the call to `beginTransaction`, `main2`, executing the while loop, `main3`, executing the conditional statement, and `main4`, denoting the end of the execution – which we will call the *methods* of the environment. To invoke these different methods, we use the relation `call1` as a stack that may be modified by these operations. For example, operation `main1` calls the `BT` operation by putting `cBT` on the top of the stack, followed by `cMain2`, that will invoke operation `main2`. In B notation, this is written as the substitution `call1 := call1 \/ {(card(call1)+1|->cBT),(card(call1)|->cMain2)}`.

```
REFINEMENT
   Transaction2
REFINES
```

```
   Transaction
SETS listop = {cBT,cCT,cAT,cModify,cMain1,cMain2,cMain3,cMain4}
VARIABLES
   BufferFree1,TrDepth1,call1,ii,mm
INVARIANT
  call1 : NAT <-> listop & ii : NAT & mm : NAT
  & BufferFree = BufferFree1 & TrDepth = TrDepth1
INITIALISATION
   BufferFree1 := 10 || TrDepth1 := FALSE ||
   call1 := {(1|->cMain1)} || mm := 0 || ii :=16
OPERATIONS
   BT = SELECT TrDepth1 = FALSE & (BufferFree1 = 10) &
             call1(card(call1)) = cBT
        THEN TrDepth1 := TRUE ||
             call1 := call1 - {(card(call1))|-> cBT}
        END;

   CT = SELECT (TrDepth1 = TRUE) & (BufferFree1 > 0) &
               call1(card(call1)) = cCT
        THEN TrDepth1 := FALSE || BufferFree1 := 10 ||
             call1 := call1 - {(card(call1))|-> cCT}
        END;

   AT = SELECT (TrDepth1 = TRUE) & BufferFree1 = 0 &
               call1(card(call1)) = cAT
        THEN TrDepth1 := FALSE || BufferFree1 := 10 ||
             call1 := call1 - {(card(call1))|-> cAT}
        END;

   Modify = SELECT (TrDepth1 = TRUE) & BufferFree1 > 0 &
                    call1(card(call1)) = cModify
            THEN BufferFree1 := BufferFree1 - 1 ||
                 call1 := call1 - {(card(call1))|-> cModify}
            END;

   main1 = SELECT call1 = {(1|->cMain1)}
           THEN call1 := call1 \/ {(card(call1)+1|->cBT),(card(call1)|->cMain2)
                 - {(card(call1)|->cMain1)}
           END;

   main2 = SELECT call1(card(call1)) = cMain2 & card(call1)=1
           THEN IF (mm <= ii) & (mm < 10)
```

```
              THEN call1 := call1 \/ {(card(call1)+1|->cModify)}
               || mm := mm + 1
               ELSE call1 := call1 \/ {(card(call1)|->cMain3)}
                    -{(card(call1)|->cMain2)}
       END
          END;

   main3 = SELECT call1(card(call1)) = cMain3 & card(call1)=1
           THEN  IF (mm = 10)
         THEN call1 := call1 \/
                 {(card(call1)+1|->cAT),(card(call1)|->cMain4)}
                 - {(card(call1)|->cMain3)}
         ELSE call1 := call1 \/
                 {(card(call1)+1|->cCT),(card(call1)|->cMain4)}
                 - {(card(call1)|->cMain3)}
         END
   END;

    main4 = SELECT call1(card(call1)) = cMain4  & card(call1)=1 THEN skip END
END
```

Unité de recherche INRIA Sophia Antipolis
2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex (France)

Unité de recherche INRIA Futurs : Parc Club Orsay Université - ZAC des Vignes
4, rue Jacques Monod - 91893 ORSAY Cedex (France)
Unité de recherche INRIA Lorraine : LORIA, Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex (France)
Unité de recherche INRIA Rennes : IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex (France)
Unité de recherche INRIA Rhône-Alpes : 655, avenue de l'Europe - 38334 Montbonnot Saint-Ismier (France)
Unité de recherche INRIA Rocquencourt : Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex (France)

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-6399