JML-based Verification of Liveness Properties on a Class in Isolation
SAVCBS 2006

Julien Groslambert  Jacques Julliand  Olga Kouchnarenko

November 10-11th 2006
Portland, Oregon.

LIFC - University of Franche-Conté
Motivations

Formal Verification of Conformity between

- Requirements
- Implementation source code

Requirements

- Absence of null pointer exception.
- Class Invariance.
- Temporal behavior.

Temporal properties

+ are expressible in JML.
- need a tedious work for annotating.

Annotations for Java:

Leavens and Al.
Well tool supported.
Motivations

Formal Verification of Conformity between

- Requirements
- Implementation source code

Conformity?

Requirements

- Absence of null pointer exception.
- Class Invariance.
- Temporal behavior.

Temporal properties

- are expressible in JML.
- need a tedious work for annotating.

JML Annotations

- Leavens and Al.
- Annotations for Java.
- Well tool supported.
Motivations

Requirements
- Absence of null pointer exception.
- Class Invariance.
- Temporal behavior.

JML Annotations
- Leavens and Al.
- Annotations for Java.
- Well tool supported.

Temporal properties
+ are expressible in JML.
- need a tedious work for annotating.

⇒ Automatic annotation generation from high level temporal properties.
Motivations

Requirements

- Absence of null pointer exception.
- Class Invariance.
- Temporal behavior.

JML Annotations

- Leavens and Al.
- Annotations for Java.
- Well tool supported.

Temporal properties

- are expressible in JML.
- need a tedious work for annotating.

→ Automatic annotation generation from high level temporal properties.
Motivations

Requirements
- Absence of null pointer exception.
- Class Invariance.
- Temporal behavior.

JML Annotations
- Leavens and Al.
- Annotations for Java.
- Well tool supported.

Temporal properties
+ are expressible in JML.
- need a tedious work for annotating.

⇒ **Automatic annotation generation** from high level temporal properties.
Proposed Approach - Huisman Trentelman [AMAST’02]

1. Expressing **security properties** from requirements
2. **Translating** properties into a **annotation language** for the implementation
3. **Verifying** the properties on the code

Focus of the talk

Extension of the approach to **Liveness Properties**
Proposed Approach - Huisman Trentelman [AMAST’02]

1. Expressing **security properties** from requirements
2. Translating properties into a **annotation language** for the implementation
3. Verifying the properties on the **code**

Focus of the talk
Extension of the approach to **Liveness Properties**
Proposed Approach - Huisman Trentelman [AMAST’02]

1. Expressing security properties from requirements
2. Translating properties into a annotation language for the implementation
3. Verifying the properties on the code

Focus of the talk
Extension of the approach to Liveness Properties
Proposed Approach - Huisman Trentelman [AMAST’02]

1. Expressing **security properties** from requirements
2. **Translating** properties into a **annotation language** for the implementation
3. **Verifying** the properties on the **code**

Temporal Logic

JML

Proof of Consistency between Java and JML

Focus of the talk

Extension of the approach to **Liveness Properties**
Proposed Approach - Huisman Trentelman [AMAST’02]

1. **Expressing security properties from requirements**

2. **Translating** properties into a **annotation language** for the implementation

3. **Verifying** the properties on the **code**

Focus of the talk

**Extension of the approach to Liveness Properties**
1. Introduction

2. Verification of Liveness Properties with JML

3. The JAG Tool

4. Conclusion and Future Work
Introduction

Verification of Liveness Properties with JML

The JAG Tool

Conclusion and Future Work
Running Example: A Transaction System

Behavior

Two steps:
- Personalization.
  - storeData: fix the size of the buffer.
- Use.
  - begin a transaction.
  - write a modification.
  - commit transaction.
  - abort transaction.
Running Example: A Transaction System

Behavior

Two steps:
- **Personalization.**
  - storeData: fix the size of the buffer.
- **Use.**
  - begin a transaction.
  - write a modification.
  - commit transaction.
  - abort transaction.
Behavior

Two steps:
- Personalization.
  - storeData: fix the size of the buffer.
- Use.
  - begin a transaction.
  - write a modification.
  - commit transaction.
  - abort transaction.
Behavior

Two steps:
- Personalization.
  - storeData: fix the size of the buffer.
- Use.
  - begin a transaction.
  - write a modification.
  - commit transaction.
  - abort transaction.
Running Example: A Transaction System

Behavior

Two steps:
- Personalization.
  - storeData: fix the size of the buffer.
- Use.
  - begin a transaction.
  - write a modification.
  - commit transaction.
  - abort transaction.
Running Example: A Transaction System

Behavior

Two steps:
- **Personalization.**
  - storeData: fix the size of the buffer.
- **Use.**
  - begin a transaction.
  - write a modification.
  - commit transaction.
  - abort transaction.
Running Example: A Transaction System

**Fields**

- **perso**: boolean describing if the card is already personalized.
- **len**: Integer representing the length of the Buffer.
- **status**: byte array specifying the status of the system.
- **buffer**: byte array specifying a temporal buffer.
- **position**: integer representing the current position in the Buffer.
- **trDepth**: boolean describing if there is a current transaction.

**Example**

```java
public class Buffer {
    ...
    boolean perso = false;
    int len;
    byte [] status;
    byte [] buffer;
    int position = 0;
    boolean trDepth = false;
    ...
}
```
Running Example: A Transaction System

**Fields**

- **perso**: boolean describing if the card is already personalized.
- **len**: Integer representing the length of the Buffer.
- **status**: byte array specifying the status of the system.
- **buffer**: byte array specifying a temporal buffer.
- **position**: integer representing the current position in the Buffer.
- **trDepth**: boolean describing if there is a current transaction.

**Example**

```java
public class Buffer {
    ...
    boolean perso = false;
    int len;
    byte [] status;
    byte [] buffer;
    int position = 0;
    boolean trDepth = false;
    ...
}
```
Running Example: A Transaction System

**Fields**

- **perso**: boolean describing if the card is already personalized.
- **len**: Integer representing the length of the Buffer.
- **status**: byte array specifying the status of the system.
- **buffer**: byte array specifying a temporal buffer.
- **position**: integer representing the current position in the Buffer.
- **trDepth**: boolean describing if there is a current transaction.

**Example**

```java
public class Buffer {
    ...
    boolean perso = false;
    int len;
    byte [] status;
    byte [] buffer;
    int position = 0;
    boolean trDepth = false;
    ...
}
```
Running Example: A Transaction System

### Fields

- **perso: boolean** describing if the card is already personalized.
- **len: Integer** representing the length of the Buffer.
- **status: byte array** specifying the status of the system.
- **buffer: byte array** specifying a temporary buffer.
- **position: integer** representing the current position in the Buffer.
- **trDepth: boolean** describing if there is a current transaction.

### Example

```java
public class Buffer {
    ...
    boolean perso = false;
    int len;
    byte [] status;
    byte [] buffer;
    int position = 0;
    boolean trDepth = false;
    ...
}
```
Fields

**perso:** boolean describing if the card is already personalized.

**len:** Integer representing the length of the Buffer.

**status:** byte array specifying the status of the system.

**buffer:** byte array specifying a temporal buffer.

**position:** integer representing the current position in the Buffer.

**trDepth:** boolean describing if there is a current transaction.

Example

```java
public class Buffer {
    ... boolean perso = false;
    int len;
    byte [] status;
    byte [] buffer;
    int position = 0;
    boolean trDepth = false;
    ...
} 
```
Running Example: A Transaction System

Fields

perso: boolean describing if the card is already personalized.
len: Integer representing the length of the Buffer.
status: byte array specifying the status of the system.
buffer: byte array specifying a temporal buffer.
position: integer representing the current position in the Buffer.
trDepth: boolean describing if there is a current transaction.

Example

```java
public class Buffer {
    ...
    boolean perso = false;
    int len;
    byte [] status;
    byte [] buffer;
    int position = 0;
    boolean trDepth = false;
    ...
}
```
Running Example: A Transaction System

**Fields**

- **perso**: boolean describing if the card is already personalized.
- **len**: Integer representing the length of the Buffer.
- **status**: byte array specifying the status of the system.
- **buffer**: byte array specifying a temporal buffer.
- **position**: integer representing the current position in the Buffer.
- **trDepth**: boolean describing if there is a current transaction.

**Example**

```java
class Buffer {
    ...  
    boolean perso = false;
    int len;
    byte [] status;
    byte [] buffer;
    int position = 0;
    boolean trDepth = false;
    ...  
} 
```
Methods

- `storeData` to personalize the Transaction System.
- `begin` to start a new transaction.
- `write` to write in the current Buffer.
- `getBufferLess` to get the Buffer free place
- `commit` to valid the current transaction.
- `getStatus` to get the current status of the transaction.
- `abort` to abort the current transaction.

Example

```java
public class Buffer {
    ...
    void storeData(int l) {
        len = l;
        perso = true;
    }
    void begin() throws Exception {
        if (perso == false) {
            throw new Exception();
        }
        buffer = new byte[len];
        trDepth = true;
    }
    ...
```
Running Example: A Transaction System

**Methods**

- **storeData** to personalize the Transaction System.
- **begin** to start a new transaction.
- **write** to write in the current Buffer.
- **getBufferLess** to get the Buffer free place.
- **commit** to valid the current transaction.
- **getStatus** to get the current status of the transaction.
- **abort** to abort the current transaction.

**Example**

```java
public class Buffer {
    ...
    void storeData(int l) {
        len = l;
        perso = true;
    }
    void begin()
        throws Exception{
            if (perso == false) {
                throw new Exception();
            } 
            buffer = new byte[len];
            trDepth = true;
        }
    ...
```
### Methods

- **storeData** to personalize the Transaction System.
- **begin** to start a new transaction.
- **write** to write in the current Buffer.
- **getBufferLess** to get the Buffer free place
- **commit** to valid the current transaction.
- **getStatus** to get the current status of the transaction.
- **abort** to abort the current transaction.

### Example

```java
public class Buffer {
    ...
    void storeData(int l) {
        len = l;
        perso = true;
    }
    void begin() throws Exception {
        if (perso == false) {
            throw new Exception();
        }
        buffer = new byte[len];
        trDepth = true;
    }
    ...
}
Running Example: A Transaction System

**Methods**
- `storeData` to personalize the Transaction System.
- `begin` to start a new transaction.
- `write` to write in the current Buffer.
- `getBufferLess` to get the Buffer free place
- `commit` to valid the current transaction.
- `getStatus` to get the current status of the transaction.
- `abort` to abort the current transaction.

**Example**
```java
public class Buffer {
    ...

    void write(byte b){
        buffer[position] = b;
        position++;
    }

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

storeData to personalize the Transaction System.
begin to start a new transaction.
write to write in the current Buffer.
getBufferLess to get the Buffer free place
commit to valid the current transaction.
getStatus to get the current status of the transaction.
abort to abort the current transaction.

Example

```java
public class Buffer {
  ...

  void write(byte b){
    buffer[position] = b;
    position++;
  }

  int getBufferLess(){
    return len - buffer.length;
  }
}```
Running Example: A Transaction System

Methods

storeData to personalize the Transaction System.
begin to start a new transaction.
write to write in the current Buffer.
getBufferLess to get the Buffer free place
commit to valid the current transaction.
getStatus to get the current status of the transaction.
abort to abort the current transaction.

Example

```java
public class Buffer {
    ...

    void commit()
    {
        status = buffer;
        position = 0;
        trDepth = false;
    }

    byte [] getStatus()
    {
        return status;
    }

    void abort()
    {
        position = 0;
        trDepth = false;
    }
```
Methods

storeData to personalize the Transaction System.

begin to start a new transaction.

write to write in the current Buffer.

getBufferLess to get the Buffer free place

commit to valid the current transaction.

getStatus to get the current status of the transaction.

abort to abort the current transaction.

Example

```java
class Buffer {
    ...

    void commit() {
        status = buffer;
        position = 0;
        trDepth = false;
    }

    byte[] getStatus() {
        return status;
    }

    void abort() {
        position = 0;
        trDepth = false;
    }
}
```
Running Example: A Transaction System

Methods

storeData to personalize the Transaction System.
begin to start a new transaction.
write to write in the current Buffer.
getBufferLess to get the Buffer free place
commit to valid the current transaction.
getStatus to get the current status of the transaction.
abort to abort the current transaction.

Example

```java
public class Buffer {

    ...  

    void commit(){
        status = buffer;
        position = 0;
        trDepth = false;
    }

    byte [] getStatus(){
        return status;
    }

    void abort(){
        position = 0;
        trDepth = false;
    }

```
public class Buffer {
    ...
    boolean perso = false;
    int len;
    byte [] status;
    byte [] buffer;
    int position = 0;
    boolean trDepth = false;
JML Class Specification

JML Annotations

Main JML Annotations.

1. Class invariant specification.
2. History constraint specification.

Example

```java
public class Buffer {
    ...
    boolean perso = false;
    int len;
    byte [] status;
    byte [] buffer;
    int position = 0;
    boolean trDepth = false;
}
```
JML Class Specification

JML Annotations

Main JML Annotations.
1. Class invariant specification.
2. History constraint specification.

Example

```java
public class Buffer {
    //@ invariant position >= 0;
    ...
    boolean perso = false;
    int len;
    byte [] status;
    byte [] buffer;
    int position = 0;
    boolean trDepth = false;
```
JML Class Specification

JML Annotations

Main JML Annotations.
1. Class invariant specification.
2. History constraint specification.

Invariant

Properties that have to be true in all \textit{visible} states:

- Before invocation of a method
- After invocation of a method

Example

```java
public class Buffer {
    //@ invariant position >= 0;
    ...
    boolean perso = false;
    int len;
    byte [] status;
    byte [] buffer;
    int position = 0;
    boolean trDepth = false;
```
Main JML Annotations.

1. Class invariant specification.
2. History constraint specification.

Example

```java
public class Buffer {
    //@ invariant position >= 0;
    ... 
    boolean perso = false;
    int len;
    byte [] status;
    byte [] buffer;
    int position = 0;
    boolean trDepth = false;
}
```
JML Annotations

Main JML Annotations.

1. Class invariant specification.
2. History constraint specification.

Constraint

Property linking two visible states.

- old keyword.
- for keyword.

Example

```java
public class Buffer {
    //@ invariant position >= 0;
    /*@ constraint perso ==> @ len == \old(len);
    @*/
    ...
    boolean perso = false;
    int len;
    byte [] status;
    byte [] buffer;
    int position = 0;
    boolean trDepth = false;
}```
JML Class Specification

JML Annotations

Main JML Annotations.
1. Class invariant specification.
2. History constraint specification.

Constraint

Property linking two visible states.
- `old` keyword.
- `for` keyword.

Example

```java
public class Buffer {
    //@ invariant position >= 0;
    //@ constraint perso ==> len == \old(len);
    @*
    //@ constraint
    @ position >= \old(position);
    @ for write;
    @*/
    ...
    boolean perso = false;
    int len;
    byte [] status;
    byte [] buffer;
    int position = 0;
    boolean trDepth = false;
}```
JML Class Specification

JML Annotations
Main JML Annotations.
1. Class invariant specification.
2. History constraint specification.

Example
void begin() throws Exception{
    ...
}

Method Specification
1. Precondition.
2. Postcondition.
3. Exceptional Postcondition.
4. Frame Condition.
JML Class Specification

JML Annotations

Main JML Annotations.
1. Class invariant specification.
2. History constraint specification.

Example
/*@ 
@ requires trDepth == false;
@ requires perso == true;
@*/
void begin() throws Exception{
    ...
}
JML Class Specification

JML Annotations

Main JML Annotations.
1. Class invariant specification.
2. History constraint specification.

Example

/*@ 
@ requires trDepth == false;
@ requires perso == true;
@ ensures trDepth == true;
@*/
void begin() throws Exception{
    ...
}

Method Specification

1. Precondition.
2. Postcondition.
3. Exceptional Postcondition.
4. Frame Condition.
JML Class Specification

JML Annotations

Main JML Annotations.
1. Class invariant specification.
2. History constraint specification.

Method Specification

1. Precondition.
2. Postcondition.
3. Exceptional Postcondition.
4. Frame Condition.

Example

```java
/*@ normal_behavior
@ requires trDepth == false;
@ requires perso == true;
@ ensures trDepth == true;
@ also
@ exceptional_behavior
@ requires perso == false;
@ signals (Exception e) true;
@*/
void begin() throws Exception{
    ...
}
```
JML Class Specification

JML Annotations

Main JML Annotations.
1. Class invariant specification.
2. History constraint specification.

Method Specification

1. Precondition.
2. Postcondition.
3. Exceptional Postcondition.
4. Frame Condition.

Example

/*@ normal_behavior
@ requires trDepth == false;
@ requires perso == true;
@ assignable buffer;
@ ensures trDepth == true;
@ also
@ exceptional_behavior
@ requires perso == false;
@ assignable \nothing;
@ signals (Exception e) true;
@*/
void begin() throws Exception{
    ...
}
Modular Reasoning

Reasoning Modularly consists in
1. Establishing a property of a class in isolation assuming some hypothesis of the program using the class.
2. Verifying the hypothesis on the program.

Java Class

Java Program
Using the class
Reasoning Modularly consists in

1. Establishing a property of a class in isolation assuming some hypothesis of the program using the class.

2. Verifying the hypothesis on the program.

<table>
<thead>
<tr>
<th>Java Class</th>
<th>Java Program Using the class</th>
</tr>
</thead>
</table>

Verification on isolation  Hypothesis
Modular Reasoning

Reasoning Modularly consists in

1. Establishing a property of a class in isolation assuming some hypothesis of the program using the class.
2. Verifying the hypothesis on the program.

Java Class

Java Program
Using
the class

Verification of the Hypothesis
Modular Reasoning - Example: Method Correctness

Design by contract approach.

Java Class Contract
- Assumes the Precondition.
- Establishes the Postcondition.

Java Program using the class
- Assumes the Postcondition.
- Establishes the Precondition.

Example
```java
/*@ requires trDepth;
@ requires buffer != null;
@ requires position > 0;
@ requires position < buffer.length;
@ ensures position == \old(position)+1;
@*/
void write(byte b){
    buffer[position] = b;
    position++;}
```

Example
```java
a.storeDate(4);
// assert precondition
a.begin();
// assume postcondition
a.write(7);
// assume postcondition
```
Temporal Properties

Examples of Temporal Requirements for the Buffer

1. The application can be personalized only once.
2. The status is always the same unless a commit happens.
3. A begin must inevitably been followed by a commit or an abort.
Examples of Temporal Requirements for the Buffer

1. The application can be personalized only once.
2. The status is always the same unless a commit happens.
3. A begin must inevitably been followed by a commit or an abort.
Temporal Properties

Examples of Temporal Requirements for the Buffer

1. The application can be personalized only once.
2. The status is always the same unless a commit happens.
3. A begin must inevitably been followed by a commit or an abort.
Introduction

Verification of Liveness Properties with JML

The JAG Tool

Conclusion and Future Work
1. Introduction

2. Verification of Liveness Properties with JML

3. The JAG Tool

4. Conclusion and Future Work
Introduction

Verification of Liveness Properties with JML

The JAG Tool

Conclusion and Future Work
1. Introduction

2. Verification of Liveness Properties with JML

3. The JAG Tool

4. Conclusion and Future Work
1. Introduction

2. Verification of Liveness Properties with JML

3. The JAG Tool

4. Conclusion and Future Work
**Definition (Loop Primitive)**

Loop$(Q)$ A state where $Q$ is satisfied must be inevitably followed by a state where $Q$ is not satisfied.

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

**Illustration**

![Diagram](image)

**Example**

A `begin` must inevitably been followed by a `commit` or an `abort`.

Loop$(\text{TrDepth} == \text{true})$
Modular Reasoning - Application to liveness properties

Java Class contract
- Assumes a Progress Hypothesis
- Establishes the Liveness on a class in isolation.

Java Program Contract
- Assumes the liveness on a class in Isolation.
- Establishes a Progress Hypothesis.

Satisfaction of the liveness by the whole program.
Progress Hypothesis

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

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

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

Progress methods are infinitely often called.
The program stay in a state satisfying \( \neg Q \).
Variant Introduction

Need a variant $V$ like a proof termination.

- Given by the user
- Well founded $\Rightarrow$ Expression from a subset of the class variables to the positive integers
- must decrease for each method invocation until $Q$

Example

Loop($TrDepth$) The variant $V$ is $getBufferLess()$. 
Annotations

//@ invariant V \geq 0;
//@ constraint Q ==> V < \old(V) for M;
//@ constraint Q ==> V <= \old(V);
//@ invariant Q ==> \bigvee_{m \in M} requires(m);
//@ invariant Q ==> \bigwedge_{m \in M_c} (requires(m) ==> !diverges(m));

Example (Loop(TrDepth))
Well-foundation of $V$

//@ invariant $V \geq 0$; 
//@ constraint $Q \implies V < old(V)$ for $M$;  
//@ constraint $Q \implies V \leq old(V)$;  
//@ invariant $Q \implies \bigvee_{m \in M} \text{requires}(m)$  
//@ invariant $Q \implies \bigwedge_{m \in M_c} (\text{requires}(m) \implies !\text{diverges}(m))$;

Example (Loop(TrDepth))
//@ invariant getBufferLess() $\geq 0$
Each progress method decreases $V$

//@ invariant $V \geq 0$; \hspace{1cm} (A_1)
//@ constraint $Q \implies V < \\text{old}(V)$ for $M$; \hspace{1cm} (A_2)
//@ constraint $Q \implies V \leq \\text{old}(V)$; \hspace{1cm} (A_3)
//@ invariant $Q \implies \forall m \in M \text{ requires}(m)$ \hspace{1cm} (A_4)
//@ invariant $Q \implies \bigwedge_{m \in M_c} (\text{requires}(m) \implies \neg \text{diverges}(m))$; \hspace{1cm} (A_5)

Example (Loop(TrDepth))

/*@ constraint trDepth 
\implies \text{getBufferLess}() < \\text{old}(\text{getBufferLess}()) \text{ for storeData, begin, abort, commit, write;}
Each method does not increase $V$

```plaintext
//@ invariant $V \geq 0$;  
//@ constraint $Q \implies V < \ \old(V)$ for $M$;  
//@ constraint $Q \implies V \leq \old(V)$ ;  
//@ invariant $Q \implies \bigvee_{m \in M} \text{requires}(m)$  
//@ invariant $Q \implies \bigwedge_{m \in M_C} (\text{requires}(m) \implies \neg \text{diverges}(m))$;
```

Example (Loop(TrDepth))

```plaintext
//@ constraint trDepth  
@ ==> getBufferLess() \leq \old(getBufferLess()) ;
```
No dead-lock for the class

//@ invariant V >= 0; (A_1)
//@ constraint Q ==> V < \ old(V) for M; (A_2)
//@ constraint Q ==> V <= \ old(V); (A_3)
//@ invariant Q ==> \( \forall m \in M \text{ requires}(m) \) (A_4)
//@ invariant Q ==> \( \forall m \in M_C \text{ (requires}(m) \implies !\text{diverges}(m)) \); (A_5)

Example (Loop(TrDepth))

/*@ invariant trDepth ==> ( perso == false || @ (trDepth == false && perso == true) || @ (trDepth == true && perso == true && position < len)) */
No divergence for the class

//@ invariant \( V \geq 0 \); \hspace{1cm} (A_1)
//@ constraint \( Q \implies V < \ \text{old}(V) \) for \( M \); \hspace{1cm} (A_2)
//@ constraint \( Q \implies V \leq \ \text{old}(V) \); \hspace{1cm} (A_3)
//@ invariant \( Q \implies \bigvee_{m \in M} \text{requires}(m) \) \hspace{1cm} (A_4)
//@ invariant \( Q \implies \bigwedge_{m \in M_c} (\text{requires}(m) \implies \neg \text{diverges}(m)) \); \hspace{1cm} (A_5)

Example (Loop(TrDepth))

obvious Annotation
Soundness of the method

Theorem

If $C : A_{1-5} \land PH(Q, M)$ then $\text{Loop}(Q)$. 
Soundness of the method

**Theorem**

If $C : A_{1-5} \land PH(Q, M)$ then $\text{Loop}(Q)$.

**Intuition.**

- $A_4$. No dead-lock for the class
- $A_5$. No divergence for the class
- $A_2$. Each progress method decreases $V$
- $A_3$. Each method does not increase $V$
- Decrease of the variant
- $A_1$. Well-foundation of $V$
Soundness of the method

**Theorem**

If \( C : A_{1-5} \land PH(Q, M) \) then \( Loop(Q) \).

**Intuition.**

- \( A_4 \). No dead-lock for the class
- \( A_5 \). No divergence for the class
- \( A_2 \). Each progress method decreases \( V \)
- \( A_3 \). Each method does not increase \( V \)
- Decrease of the variant
- \( A_1 \). Well-foundation of \( V \)
Soundness of the method

Theorem

If \( C : A_{1-5} \land PH(Q, M) \) then \( Loop(Q) \).

Intuition.

- \( A_4 \). No dead-lock for the class
- \( A_5 \). No divergence for the class
- \( A_2 \). Each progress method decreases \( V \)
- \( A_3 \). Each method does not increase \( V \)
- Decrease of the variant
- \( A_1 \). Well-foundation of \( V \)
Soundness of the method

Theorem

If \( C : A_{1-5} \land PH(Q, M) \) then \( Loop(Q) \).

Intuition.

- \( A_4 \). No dead-lock for the class
- \( A_5 \). No divergence for the class
- \( A_2 \). Each progress method decreases \( V \)
- \( A_3 \). Each method does not increase \( V \)
- Decrease of the variant
- \( A_1 \). Well-foundation of \( V \)
Soundness of the method

Theorem

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

Intuition.

- \( A_4 \). No dead-lock for the class
- \( A_5 \). No divergence for the class
- \( A_2 \). Each progress method decreases \( V \)
- \( A_3 \). Each method does not increase \( V \)
- Decrease of the variant
- \( A_1 \). Well-foundation of \( V \)
Soundness of the method

**Theorem**

If \( C : A_{1-5} \land PH(Q, M) \) then \( Loop(Q) \).

**Intuition.**

- \( A_4 \). No dead-lock for the class
- \( A_5 \). No divergence for the class
- \( A_2 \). Each progress method decreases \( V \)
- \( A_3 \). Each method does not increase \( V \)
- Decrease of the variant
- \( A_1 \). Well-foundation of \( V \)
Soundness of the method

**Theorem**

If \( C : A_{1-5} \land PH(Q, M) \) then \( Loop(Q) \).

**Proof.**

\[ PH \quad C :: A_{1-5} \quad Loop(Q) \]
Soundness of the method

Theorem

If \( C : A_{15} \land PH(Q, M) \) then \( Loop(Q) \).

Proof.

\[
\begin{align*}
\text{PH} & \quad C :: A_{15} & \quad \text{Loop}(Q) \\
\text{Trace Semantics} & \quad \Sigma_{PH} & \quad \Sigma_{A_{15}} & \quad \Sigma_{Loop(Q)}
\end{align*}
\]
Soundness of the method

**Theorem**

\[
\text{If } C : \mathcal{A}_{1-5} \land \text{PH}(Q, M) \text{ then Loop}(Q).
\]

**Proof.**

\[
\text{PH} \land C :: \mathcal{A}_{1-5} \implies \text{Loop}(Q)
\]

---

Trace Semantics

\[
\Sigma_{PH} \cap \Sigma_{\mathcal{A}_{1-5}} \subseteq \Sigma_{\text{Loop}(Q)}
\]
1 Introduction

2 Verification of Liveness Properties with JML

3 The JAG Tool

4 Conclusion and Future Work
1 Introduction

2 Verification of Liveness Properties with JML

3 The JAG Tool

4 Conclusion and Future Work
Introduction

Verification of Liveness Properties with JML

The JAG Tool

Conclusion and Future Work
JAG: General Approach

- JAG 0.1 input: JTPL Properties [Huisman Trentelman - AMAST’02].
- Generates JML annotations ensuring the JTPL properties.
- Annotations traceability
### Example (Unique personalization)

```java
after storeData normal
  always (perso == true
          and storeData not enabled)
```

### Example (Command not enabled before personnalization)

```java
always (begin not enabled
        and commit not enabled)
  unless storeData normal
```

### Example (Begin eventually followed by Commit or Abort)

```java
after begin normal
  (eventually commit called ),
  abort called)
```
Example (Unique personalization)

\texttt{after \: storeData \: normal}
\begin{itemize}
  \item \texttt{always} (perso == true \\
  \hspace{1cm} \texttt{and} \: storeData \: \texttt{not enabled})
\end{itemize}

Example (Command not enabled before personnalization)

\begin{itemize}
  \item \texttt{always} \: (begin \: \texttt{not enabled} \\
  \hspace{1cm} \texttt{and} \: commit \: \texttt{not enabled})
  \item \texttt{unless} \: storeData \: \texttt{normal}
\end{itemize}

Example (Begin eventually followed by Commit or Abort)

\begin{itemize}
  \item \texttt{after begin normal} \\
  \hspace{1cm} (eventually commit called \\
  \hspace{1cm} abort called)
\end{itemize}
**Example (JTPL Property)**

```plaintext
after begin normal
  (eventually commit called),
  abort called)
```

**Annotation Generation**

- Declaration of a ghost variable.
- Assignment of the ghost variable.
- Loop(Witness).

**Example (Java Code)**

```java
public class Buffer {
  //@ ghost witness = false;
  boolean perso = false;
  ...
  void begin()
  ...
  //@ set witness = true;
  }
  void commit()
  //@ set witness = false;
  ...
  }
  void abort()
  //@ set witness = false;
  ...
  }
```
Example (JTPL Property)

```java
after begin normal
  (eventually commit called),
  abort called)
```

Example (Java Code)

```java
public class Buffer {
  //@ ghost witness = false;
  boolean perso = false;
  ...
  void begin(){
    ...
    //@ set witness = true;
  }
  void commit(){
    //@ set witness = false;
    ...
  }
  void abort(){
    //@ set witness = false;
    ...
  }
}
```
### Example (JTPL Property)

```
after begin normal
  (eventually commit called),
  abort called)
```

### Annotation Generation
- Declaration of a ghost variable.
- Assignment of the ghost variable.
- Loop(Witness).

### Example (Java Code)

```java
public class Buffer {
  //@ ghost witness = false;
  boolean perso = false;
  ...
  void begin(){
    ...
    //@ set witness = true;
  }
  void commit(){
    //@ set witness = false;
    ...
  }
  void abort(){
    //@ set witness = false;
    ...
  }
```

Example (JTPL Property)

\textbf{after} begin \textbf{normal} (\textbf{eventually} commit \textbf{called}), abort \textbf{called})

Example (Java Code)

```java
public class Buffer {
    //@ ghost witness = false;
    boolean perso = false;
    ...
    void begin(){
        ...
        //@ set witness = true;
    }
    void commit(){
        //@ set witness = false;
        ...
    }
    void abort(){
        //@ set witness = false;
        ...
    }
}
```

Annotation Generation

- Declaration of a ghost variable.
- Assignment of the ghost variable.
- Loop(Witness).
Feature of JAG 0.1

- Automatic generation of ghost variables for observing events and states.
- Automatic generation of invariants for safety properties [Huisman Trentelman AMAST’02]
- All liveness formulae of the language are rewrited in Loop primitive.
- Generation of standard JML file
  - Compatibility with all other JML tools
Feature of JAG 0.1

- Automatic generation of ghost variables for observing events and states.
- Automatic generation of invariants for safety properties [Huisman Trentelman AMAST’02]
- All liveness formulae of the language are rewritten in Loop primitive.
- Generation of standard JML file
  => Compatibility with all other JML tools
  - Runtime Verification (jmlc - Iowa State University)
  - Symbolic Animation, Test Generation (JML-TT - LIFC, Tobias Jartega - LSR)
  - JML annotations consistency (JML2B - LIFC)
  - Consistency of JML annotations with Java code
    - ESC-Java
    - Krakatoa - LRI
    - Jack - INRIA Sophia
Feature of JAG 0.1

- Automatic generation of ghost variables for observing events and states.
- Automatic generation of invariants for safety properties [Huisman Trentelman AMAST’02]
- All liveness formulae of the language are rewrited in Loop primitive.

- Generation of standard JML file
  - Compatibility with all other JML tools
    - Runtime Verification (jmlc - Iowa State University)
    - Symbolic Animation, Test Generation (JML-TT - LIFC, Tobias Jartege - LSR)
    - JML annotations consistency (JML2B - LIFC)
    - Consistency of JML annotations with Java code
      - ESC-Java
      - Krakatoa - LRI
      - Jack - INRIA Sophia
Automatic generation of ghost variables for observing events and states.

Automatic generation of invariants for safety properties [Huisman Trentelman AMAST’02]

All liveness formulae of the language are rewritten in Loop primitive.

Generation of standard JML file
⇒ Compatibility with all other JML tools
  ⇒ Runtime Verification (jmlc - Iowa State University)
  ⇒ Symbolic Animation, Test Generation (JML-TT - LIFC, Tobias, Jartege - LSR)
  ⇒ JML annotations consistency (JML2B - LIFC)
  ⇒ Consistency of JML annotations with Java code
    ⇒ ESC-Java
    ⇒ Krakatoa - LRI
    ⇒ Jack - INRIA Sophia
Feature of JAG 0.1

- Automatic generation of ghost variables for observing events and states.
- Automatic generation of invariants for safety properties [Huisman Trentelman AMAST’02]
- All liveness formulae of the language are rewritten in Loop primitive.
- Generation of standard JML file
  - Compatibility with all other JML tools
    - Runtime Verification (jmlc - Iowa State University)
    - Symbolic Animation, Test Generation (JML-TT - LIFC, Tobias, Jartege - LSR)
    - JML annotations consistency (JML2B - LIFC)
    - Consistency of JML annotations with Java code
      - ESC-Java
      - Krakatoa - LRI
      - Jack - INRIA Sophia
Feature of JAG 0.1

- Automatic generation of ghost variables for observing events and states.
- Automatic generation of invariants for safety properties [Huisman Trentelman AMAST’02]
- All liveness formulae of the language are rewrittend in Loop primitive.
- Generation of standard JML file
  - Compatibility with all other JML tools
    - Runtime Verification (jmlc - Iowa State University)
    - Symbolic Animation, Test Generation (JML-TT - LIFC, Tobias, Jartege - LSR)
    - JML annotations consistency (JML2B - LIFC)
    - Consistency of JML annotations with Java code
      - ESC-Java
      - Krakatoa - LRI
      - Jack - INRIA Sophia
Feature of JAG 0.1

- Automatic generation of ghost variables for observing events and states.
- Automatic generation of invariants for safety properties [Huisman Trentelman AMAST’02]
- All liveness formulae of the language are rewritten in Loop primitive.
- Generation of standard JML file
  ⇒ Compatibility with all other JML tools
    - Runtime Verification (jmlc - Iowa State University)
    - Symbolic Animation, Test Generation (JML-TT - LIFC, Tobias, Jartege - LSR)
    - JML annotations consistency (JML2B - LIFC)
  Consistency of JML annotations with Java code
    - ESC-Java
    - Krakatoa - LRI
    - Jack - INRIA Sophia
Feature of JAG 0.1

- Automatic generation of ghost variables for observing events and states.
- Automatic generation of invariants for safety properties [Huisman Trentelman AMAST’02]
- All liveness formulae of the language are rewrited in Loop primitive.
- Generation of standard JML file
  - Compatibility with all other JML tools
    - Runtime Verification (jmlc - Iowa State University)
    - Symbolic Animation, Test Generation (JML-TT - LIFC, Tobias, Jartege - LSR)
    - JML annotations consistency (JML2B - LIFC)
    - Consistency of JML annotations with Java code
      - ESC-Java
      - Krakatoa - LRI
      - Jack - INRIA Sophia
### Case study: Demoney

- JavaCard Electronic Purse
- Over 500 lines of JML

### Case study: Demoney

- Annotation Generation with JAG on the JML model and proof with JML2B [B’07].
- Generation of tests with JML-TT and verification of the annotations generated by JAG at the runtime [FATES’06]
1 Introduction

2 Verification of Liveness Properties with JML

3 The JAG Tool

4 Conclusion and Future Work
Introduction

Verification of Liveness Properties with JML

The JAG Tool

Conclusion and Future Work
Conclusion

- **Sound method** for verifying liveness on a class in isolation with JML.
- Trace-based semantics framework for reasoning about Java/JML.
- Reusable Liveness Primitive Operator.
- Tool supported: JAG.
- Generation of standard JML annotations.
- Experiment on a Java Card Application.
Conclusion

- **Sound method** for verifying **liveness** on a class in isolation with JML.
- **Trace-based semantics** framework for reasoning about Java/JML.
  - Reusable Liveness Primitive Operator.
  - Tool supported: JAG.
  - Generation of standard JML annotations.
  - Experiment on a Java Card Application.
Conclusion

- **Sound method** for verifying liveness on a class in isolation with JML.
- **Trace-based semantics** framework for reasoning about Java/JML.
- **Reusable** Liveness Primitive Operator.
- Tool supported: JAG.
- Generation of **standard JML annotations**.
- Experiment on a Java Card Application.
Conclusion

- **Sound method** for verifying **liveness** on a class in isolation with JML.
- **Trace-based semantics** framework for reasoning about Java/JML.
- **Reusable** Liveness Primitive Operator.
- Tool supported: **JAG**.
- Generation of **standard JML annotations**.
- Experiment on a Java Card Application.
Sound method for verifying liveness on a class in isolation with JML.

Trace-based semantics framework for reasoning about Java/JML.

Reusable Liveness Primitive Operator.

Tool supported: JAG.

Generation of standard JML annotations.

Experiment on a Java Card Application.
Sound method for verifying liveness on a class in isolation with JML.

Trace-based semantics framework for reasoning about Java/JML.

Reusable Liveness Primitive Operator.

Tool supported: JAG.

Generation of standard JML annotations.

Experiment on a Java Card Application.
Work in Progress

- Verification of the **progress** of the program.
- Extension to other input/output languages
  - SPEC♯
  - B [B’07]
  - Aspect J
- Extension to other formalisms
  - LTL
  - Transition Diagram
  - Regular Expression
  - CaRet
- Temporal Properties conformance testing [FATES’06]
- Collaboration of proof and test techniques [JML-TT]
  - Generating specific test cases when the proof fails.
  - Trying to find a counter-example.
Work in Progress

- Verification of the **progress** of the program.
- Extension to other input/output languages
  - SPEC#
  - B [B’07]
  - Aspect J
- Extension to other formalisms
  - LTL
  - Transition Diagram
  - Regular Expression
  - CaRet
- Temporal Properties conformance testing [FATES’06]
- Collaboration of proof and test techniques [JML-TT]
  - Generating specific test cases when the proof fails.
  - Trying to find a counter-example.
Work in Progress

- Verification of the progress of the program.
- Extension to other input/output languages
  - SPEC#
  - B [B’07]
  - Aspect J
- Extension to other formalisms
  - LTL
  - Transition Diagram
  - Regular Expression
  - CaRet
- Temporal Properties conformance testing [FATES’06]
- Collaboration of proof and test techniques [JML-TT]
  - Generating specific test cases when the proof fails.
  - Trying to find a counter-example.
Work in Progress

- Verification of the progress of the program.
- Extension to other input/output languages
  - SPEC#
  - B [B’07]
  - Aspect J
- Extension to other formalisms
  - LTL
  - Transition Diagram
  - Regular Expression
  - CaRet
- Temporal Properties conformance testing [FATES’06]
  - Collaboration of proof and test techniques [JML-TT]
    - Generating specific test cases when the proof fails.
    - Trying to find a counter-example.
Work in Progress

- Verification of the progress of the program.
- Extension to other input/output languages
  - SPEC#
  - B [B’07]
  - Aspect J
- Extension to other formalisms
  - LTL
  - Transition Diagram
  - Regular Expression
  - CaRet
- Temporal Properties conformance testing [FATES’06]
- Collaboration of proof and test techniques [JML-TT]
  - Generating specific test cases when the proof fails.
  ⇒ Trying to find a counter-example.