Formal Verification of Memory Performance Contracts

Christian Engel

Universität Karlsruhe (TH)

6th International KeY Symposium
Outline

Motivation

Realtime Java

JML WCMU Specifications

Java DL and Memory Usage

Demo
Is formal verification of performance constraints really necessary?
Motivation

Is formal verification of performance constraints really necessary?

No
Motivation

Is formal verification of performance constraints really necessary?

No
Usually bad performance is not an issue of software correctness.
Motivation

Is formal verification of performance constraints really necessary?

No
Usually bad performance is not an issue of software correctness.

But . . .
Real-time applications have to meet certain performance constraints, otherwise they are erroneous.
One obstacle for writing real-time Java applications: Garbage Collection.

- Immortal Memory
- Scoped Memory
- Both are not subject to garbage collection and Threads
- Realtime Thread
- No-Heap Realtime Thread
One obstacle for writing real-time Java applications: Garbage Collection
One obstacle for writing real-time Java applications: Garbage Collection

New Types of Heap Memory . . .

▶ Immortal Memory
One obstacle for writing real-time Java applications: Garbage Collection

New Types of Heap Memory . . .
- Immortal Memory
- Scoped Memory
One obstacle for writing real-time Java applications: Garbage Collection

New Types of Heap Memory . . .

- Immortal Memory
- Scoped Memory
- Both are not subject to garbage collection
One obstacle for writing real-time Java applications: Garbage Collection

New Types of Heap Memory . . .
- Immortal Memory
- Scoped Memory
- Both are not subject to garbage collection

. . . and Threads
- Realtime Thread
RTSJ – The Real-Time Specification for Java

One obstacle for writing real-time Java applications: Garbage Collection

New Types of Heap Memory . . .
  ▶ Immortal Memory
  ▶ Scoped Memory
  ▶ Both are not subject to garbage collection

...and Threads
  ▶ Realtime Thread
  ▶ No-Heap Realtime Thread
Worst Case Execution Time (WCET)
Worst Case Execution Time (WCET)

- `duration` clause: part of the method contract
Worst Case Execution Time (WCET)

- `duration` clause: part of the method contract
- `\duration` function: `\duration(o.m())`
JML Performance Specifications

Worst Case Execution Time (WCET)
- `duration` clause: part of the method contract
- \`duration` function: `\duration(o.m())`

Worst Case Memory Usage (WCMU)
- `working_space` clause:
  - part of the method contract
  - specifies the WCMU of a method
Worst Case Execution Time (WCET)

- `duration` clause: part of the method contract
- `\duration` function: `\duration(o.m())`

Worst Case Memory Usage (WCMU)

- `working_space` clause:
  - part of the method contract
  - specifies the WCMU of a method
- `\working_space` function: `\working_space(o.m())`
Worst Case Execution Time (WCET)

- `duration` clause: part of the method contract
- `\duration` function: `\duration(o.m())`

Worst Case Memory Usage (WCMU)

- `working_space` clause:
  - part of the method contract
  - specifies the WCMU of a method
- `\working_space` function: `\working_space(o.m())`
- `\space` function: `\space(new int[3])`
Shortcomings of JML Memory Specs

This specification can be incorrect:

```java
static SomeClass instance;

/*@ working_space (clear()) +
working_space (getInstance()); @*/

public SomeClass freshInstance(){
    clear();
    return getInstance();
}
```

Verification of Performance Contracts 14.06.2007 6 / 16
Shortcomings of JML Memory Specs

This specification can be incorrect:

```java
static SomeClass instance;

/*@ working_space (clear()) + @*/
public SomeClass freshInstance(){
    clear();
    return getInstance();
}
```

- **working_space** clauses are evaluated in the post state.
- no access to intermediate program states in **working_space** expressions
Shortcomings of JML Memory Specs

This specification can be incorrect:

```java
static SomeClass instance;

public static clear(){ instance = null; }

public static getInstance(){
    if(instance==null) instance = new SomeClass();
    return instance;
}
```

- **JAVA + JML**

- **working_space** clauses are evaluated in the post state.
- no access to intermediate program states in \( working_space \) expressions
Shortcomings of JML Memory Specs

This specification can be incorrect:

```java
static SomeClass instance;

/*@ working_space \working_space(clear()) +
@ \working_space(getInstance()); */
public SomeClass freshInstance(){
    clear();
    return getInstance();
}
```

- `working_space` clauses are evaluated in the post state.
- no access to intermediate program states in `\working_space` expressions
Shortcomings of JML Memory Specs

This specification can be incorrect:

```java
static SomeClass instance;
/*@
working_space (clear()) +
@working_space (getInstance()); @*/
public SomeClass freshInstance(){
  clear(); // instance == null
  return getInstance();
}
```

- `working_space` clauses are evaluated in the post state.
- no access to intermediate program states in `working_space` expressions
Shortcomings of JML Memory Specs

This specification can be incorrect:

```java
static SomeClass instance;

/*@ working_space (clear()) + @*/

public SomeClass freshInstance(){
    clear(); // instance == null
    return getInstance(); // instance != null
}
```

- `working_space` clauses are evaluated in the post state.
- no access to intermediate program states in `working_space` expressions
Alternative Approach

State in which the target method is executed is specified within \texttt{working\_space} expressions: \texttt{working\_space(method, cond)}
Alternative Approach

State in which the target method is executed is specified within \working_space expressions: \working_space(method, cond)
\working_space(method, cond) is a rigid expression.
**Alternative Approach**

State in which the target method is executed is specified within `working_space` expressions: `working_space(method, cond)`

`working_space(method, cond)` is a rigid expression.

```java
static SomeClass instance;

/*@ working_space working_space(clear(), true) +
 @   working_space(getInstance(), instance==null);@*/
public SomeClass freshInstance(){
    clear();
    return getInstance();
}
```

---
/@ requires a!=null;
  @ working_space a.length*space(new Object()) +
  @ working_space(new ArrayStoreException(), true);
@*/

public void initArr(Object[] a){
    int i=0;
    /*@ loop_invariant i>=0;*/
    @ assignable a[*];
    @ decreasing a.length-i;
    @ working_space_single_iteration space(new Object());
    @*/
    while(i<a.length){
        a[i++] = new Object();
    }
}
Proof Obligations

--- JAVA + JML ---

/*@ public normal_behavior
  @ requires PRE;
  @ working_space S;
  @*/

public void doSth(){ ...

--- JAVA + JML ---
**Proof Obligations**

— **JAVA + JML** —

```java
/*@ public normal_behavior
  @ requires PRE;
  @ working_space S;
  @*/

public void doSth(){ ...

PRE \rightarrow \{h_{old} := h\}\langle doSth();\rangle h \leq h_{old} + S
```

---

**Idea**

Use a program variable to log the memory allocation of Java programs.

\[ PRE \rightarrow \{h_{old} := h\}\langle doSth();\rangle h \leq h_{old} + S \]
Symbolic execution of constructors increases $h$ by the heap space consumed by the created object.

**arrayCreation**

$$
\Gamma \Rightarrow \{U; h := h + space^{arr}(e, l1)\} \langle \pi AC \omega \rangle \phi, \Delta
$$

$$
\Gamma \Rightarrow \{U\} \langle \pi v=new T[l1] \rangle ; \omega \rangle \phi, \Delta
$$

**objectCreation**

$$
\Gamma \Rightarrow \{U; h := h + space_T\} \langle \pi OC \omega \rangle \phi, \Delta
$$

$$
\Gamma \Rightarrow \{U\} \langle \pi v=new T(a_1, \ldots, a_n) \rangle ; \omega \rangle \phi, \Delta
$$
Contract Rule

/*@ public normal_behavior
  @  requires Pre;
  @  ensures Post;
  @  assignable Mod;
  @  working_space S; @*/
public void m(){ ...

applyContract

Γ ⇒ {U} Pre, Δ
Γ ⇒ {U}(ws_{m()}^{nr} = \{V(\text{Mod})\} S →
{V(\text{Mod}) || h := h + ws_{m()}^{nr}})(Post → ⟨\pi \omega⟩ \phi)), Δ
Γ ⇒ {U}⟨\pi m(); \omega⟩ \phi, Δ
How \( ws^{nr} \) relates to \textbackslash working\_space

\( ws_{m}^{nr} \) is a nonrigid constant denoting the WCMU of \( m \) in a certain set of states \( S \).
How $ws^{nr}$ relates to `working_space`

$ws^{nr}_m$ is a nonrigid constant denoting the WCMU of $m$ in a certain set of states $S$.

$ws^{r}_{m, cond}$ is the Java Card DL counterpart of the JML expression `working_space(m, cond)`.
How $ws^{nr}$ relates to \texttt{working\_space}

$ws_m^{nr}$ is a nonrigid constant denoting the WCMU of $m$ in a certain set of states $S$.

$ws_m^{r, cond}$ is the \texttt{JAVA CARD DL} counterpart of the JML expression \texttt{working\_space}(m, cond).

If cond holds in every state in $S$, $\{U\} ws_m^{nr}$ cannot exceed \texttt{working\_space}(m, cond).
How $ws^{nr}$ relates to \(\text{working}\_\text{space}\)

$ws^{nr}_m$ is a nonrigid constant denoting the WCMU of $m$ in a certain set of states $S$.

$ws^{r}_{m,\text{cond}}$ is the Java Card DL counterpart of the JML expression $\text{working}\_\text{space}(m, \text{cond})$.

If $\text{cond}$ holds in every state in $S$, $\{U\} ws^{nr}_m$ cannot exceed $\text{working}\_\text{space}(m, \text{cond})$.

\[
\begin{align*}
\text{Gamma} & \Rightarrow \{U\} \text{cond}, \Delta \\
\Gamma, \{U\} ws^{nr}_m \leq ws^{r}_{m,\text{cond}} & \Rightarrow \Delta \\
\Gamma & \Rightarrow \Delta
\end{align*}
\]
Memory Usage Contract Rule

— JAVAX + JML —

/*@ public behavior
   @ requires Pre;
   @ ensures Post;
   @ working_space t;
   @*/

public int m(){ ... 

— JAVAX + JML —
Demo
Thank you for your Attention!
Questions?