Verification of Memory Performance Contracts with KeY*

Christian Engel

Universität Karlsruhe (TH), Fakultät für Informatik
Institut für theoretische Informatik
engelc@ira.uka.de

Abstract. Determining the worst case memory consumption is an important issue for real-time Java applications. This work describes a methodology for formally verifying worst case memory performance constraints and proposes extensions to Java Modeling Language (JML) facilitating better verifiability of JML performance specifications.

1 Introduction

Estimating the worst case memory usage (WCMU) of a Java application is essential for giving performance or safety guarantees. This becomes even more relevant in the context of real-time and embedded applications where the amount of memory available is inherently small and software failures caused by memory shortage are not acceptable. In the context of real-time Java [4] estimating the WCMU is especially relevant with regard to the concept of scoped memory [3], which allows defining memory areas of a fixed size (memory scopes) that are not subject to garbage collection. Since unreferenced objects in scopes are not recycled by the garbage collector, which can easily give rise to memory leaks, it would be desirable to have a means of verifying that the heap space allocated by an application does not exceed a certain upper bound, for instance the scope size.

Another field of application for WCMU analysis techniques are smart cards [7] which usually possess only several KB of RAM memory. This application scenario is particularly relevant for this work since the KeY tool forming the basis of the presented technique is targeted on the verification of programs written in JAVA CARD, a Java dialect for smart cards.

Even though the usefulness and necessity for a methodology for ensuring WCMU contraints is evident, only few theoretical works [12] [1] [9] [11] in this area exist. In practice, due to the lack of static analysis tools in this field, WCMU is often validated experimentally (by measuring the memory usage during runtime). However, this can, like testing in general, give no guarantees on the correctness of the tested WCMU estimations.

This work will elaborate on how an existing program verification system, namely the KeY [2] system, was adapted to verify memory performance contracts [14] specified

* This research was funded by the EU project DIANA (Distributed equipment Independent environment for Advanced avioNic Applications).

using the Java Modeling Language (JML) [13]. It will also propose extensions to JML that add to overcoming some known [1] shortcomings of JML's WCMU specification features.

Outline In Section 2 we give a brief summary of the KeY verification tool as much as is needed to make the following presentation self-contained. In Section 3 we first describe the memory consumption specifications of JML as they are now and then present our suggestions for improvementd. Section 4 quickly explains the general set-up of the correctness proofs resulting from our approach. Section 5 presents in detail the technical core of the proof system while Section 6 contains the results from two verification experiments that show the feasability of the proposed methodology. Sections 7 and 8 conclude with the usual wrap-up and an outlook on future work.

2 The KeY Tool

KeY [2] is a software verification tool jointly developed at the University of Karlsruhe, Chalmers University and the University of Koblenz. Based on symbolic execution and a sequent calculus for JAVA CARD DL, a dynamic logic for Java Card, it performs deductive verification of JAVA CARD programs. JAVA CARD is a sequential subset (excluding for instance floating point arithmetics) of Java.

KeY features frontends for the formal specification languages OCL [15] and JML [13], both of which are compiled to JAVA CARD DL before they can be processed in KeY. JAVA CARD DL is a dynamic logic [8] for JAVA CARD that permits to describe functional properties of JAVA CARD programs. For two formulas ϕ and ψ and a legal sequence of JAVA CARD statements $\mathtt{p}$, the formula

$$\phi \rightarrow [\mathtt{p}]\psi,$$

for instance, is equivalent to the *Hoare Triple* [10]

$$\{\phi\}p\{\psi\}$$

which is valid iff for every program state s satisfying ϕ the execution of $\mathtt{p}$ when started in s either (i) terminates in a state satisfying ψ or (ii) does not terminate. Beside this example for a partial correctness specification, total correctness is expressible with JAVA CARD DL by the diamond modality ($\langle\rangle$). Accordingly, the semantics of

$$\phi \rightarrow \langle \mathtt{p} \rangle \psi$$

is that $\mathtt{p}$ terminates when started in an arbitrary state satisfying ϕ and ψ holds in the corresponding post state.

The semantics of JAVA CARD which is needed for performing symbolic execution, is encoded in the calculus rules of the JAVA CARD DL sequent calculus. This calculus operates on proof trees whose nodes are sequents. A sequent $\Gamma \Rightarrow \Delta$, where Γ (the antecedent) and Δ (the succedent) are sets of JAVA CARD DL formulas, is valid iff the formula

$$\bigwedge_{\gamma \in \Gamma} \gamma \rightarrow \bigvee_{\delta \in \Delta} \delta \tag{1}$$

is valid. Thus the formulas in the antecedents can also be thought of as assumptions we can use to prove the antecedent to be true.

A sequent calculus rule

$$\frac{\Gamma_1 \Rightarrow \Delta_1 \quad \ldots \quad \Gamma_n \Rightarrow \Delta_n}{\Gamma \Rightarrow \Delta} \tag{2}$$

is correct if the validity of the premises (the sequents $\Gamma_i \Rightarrow \Delta_i$ with $1 \leq i \leq n$) implies the validity of the conclusion ($\Gamma \Rightarrow \Delta$). This means that a rule is basically applied bottom up: The conclusion is the sequent the rule is applied to and the premises are the result of the application. Since there is always one single premise in each rule this leads eventually to a tree shaped proof structure.

Example 21 (Calculus Rules) *The rule for treating a conjunction in the antecedent splits the proof branch it is applied to:*

$$\mathsf{andLeft}\ \frac{\Gamma \Rightarrow a,\ \Delta \quad \Gamma \Rightarrow b,\ \Delta}{\Gamma \Rightarrow a \wedge b,\ \Delta}$$

We say that a proof goal can be closed if it is an instance of an axiom (i.e. a sequent for which all of its instances are known to be valid). A sequent with $false$ ($true$) occurring on the top level of the antecedent (succedent) or the identical formula occurring in both the succedent an the antecedent is valid. Consequently open leafs in a proof tree (proof goals) of such a form can be closed.

$$\mathsf{closeByFalse}\ \frac{*}{\Gamma,\ false \Rightarrow \Delta} \qquad \mathsf{closeByTrue}\ \frac{*}{\Gamma \Rightarrow true,\ \Delta}$$

$$\mathsf{closeGoal}\ \frac{*}{\Gamma,\ \Phi \Rightarrow \Phi,\ \Delta}$$

A proof tree is closed if each of its leaves is closed indicating validity of its root node.

In general a rule is applied to an entire sequent and the result of its application is described relative to this sequent. There are however rules that can operate locally on subformulas (or subterms). We call such rules rewrite rules. For rewrite rules that are characterized by changing the sequent they are applied to only locally at one single point by replacing one subterm (or subformula) by another we use a notation in this work that differs from the notation of "normal" calculus rules. A rewrite rules replacing a formula Φ (a term s) by a formula Ψ (a term t) is written as $\frac{\Psi}{\Phi}$ and $\frac{t}{s}$ respectively. Note that Φ does not need to occur on the top level of the regarded sequent (meaning as an element of Γ or Δ) for the rewrite rule to be applied but can also be a subformula of a formula φ with $\varphi \in \Gamma \cup \Delta$. A rewrite rule $\frac{\Psi}{\Phi}$ (or $\frac{t}{s}$) is correct if for all sequents S, S', with

– Φ (or s) occurring in S, and

– S' can be derived from S by replacing an arbitrary occurrence of Φ (or s) in S by Ψ (or t),

the validity of S' implies the validity of S. Intuitively one can say a rewrite rule is correct if it describes an equivalence transformation between Φ and Ψ or s and t respectively.

Since the verification of imperative programs has to deal with side effects of program execution resulting in changes in the program state, JAVA CARD DL provides a means, called state updates, to describe those state transitions. Intuitively the semantics of an update $\mathcal{U} := \{loc_1 := val_1 || \ldots || loc_n := val_n\}$, where the terms loc_i and val_i for $1 \leq i \leq n$ are required to be side effect free, is that the values val_i are assigned to the locations loc_i (for $1 \leq i \leq n$) simultaneously. If we compose two updates $\mathcal{U}$ and $\mathcal{V}$ sequentially we write $\{\mathcal{U}; \mathcal{V}\}$

Example 22 (Sequential and Parallel Updates)
The formula $\langle$ `v=3; u=4;` $\rangle \Psi$ *is logically equivalent to* $\{v := 3; u := 4\}\Psi$ *where* `v` *is a program variable.*

The parallel update $\{a := b || b := a\}$ *swaps the values of a and b. Thus, the formula* $a = a_0 \wedge b = b_0 \rightarrow \{a := b || b := a\}(a = b_0 \wedge b = a_0)$ *is valid.*

3 JML

The Java Modeling Language (JML) [13] is a behavioral interface specification language for Java that allows, among other things, the specification of method pre- and postconditions, class invariants and assertions intermixed with Java code. KeY's JML frontend [6] covers a large fraction of JML which we will refer to in this work as KeYJML. Although the semantics of KeYJML is in principle consistent with the semantics of JML this work will propose some additions to KeYJML that modify and extend the existing JML semantics.

3.1 JML Memory Performance Specifications

Besides a variety of constructs aiming at the description of the functional behavior of Java programs, JML provides means for specifying performance properties such as worst case execution time and heap memory consumption. The worst case heap memory usage of a method under a certain precondition `PRE` is specified as part of a JML method contract [14] using the `working_space` clause:

```
/*@ public normal_behavior
  @  requires PRE;
  @  ...
  @  working_space S;
  @*/
 public void doSth(){...
```

`S` is a JML expression of type **long** that defines an upper bound on the size of heap space allocated by `doSth()` when invoked in a state satisfying `PRE`. `S` is evaluated in the post state of `doSth()` and allows (like other JML clauses evaluated in the post state) access

to the prestate by JML's `\old` construct. As already identified in [1], this restriction to two program states can be seen as a severe shortcoming of JML's memory consumption specs, on which this work will elaborate in the following.

The space allocated to an object `o` (only to the object `o` itself not including objects referenced by `o`'s attributes) can be obtained by `\space(o)`.

The JML function `\working_space` describes

> the maximum specified amount of heap space, in bytes, used by the method call or explicit constructor invocation expression that is its argument. (JML Reference Manual [13])

Example 31 *The amount of working space allocated by `m` according to its JML specification shown below can be refered to in other JML specifications with the help of the `\working_space` function.*

```
public myClass{ ...

 /*@ public normal_behavior
   @  requires a>0
   @  working_space 0;
   @ also public normal_behavior
   @  requires a<=0;
   @  working_space 8;
   @*/
 public static Object m(int a){ ...
```

The expression `\working_space(myClass.m(3))`, for instance, could be replaced by `0` since there is only one specification case, namely the first one in the above code, whose precondition (`a>0`) is `true` if `m` is called with the argument `3`.

Considering, however, the following piece of code:

```
 /*@ public normal_behavior
   @  ...
   @  ensures \result!=0;
   @  working_space
   @    \working_space(myClass.m(\result));
```

both preconditions(`a>0` and `a<=0`) can hold in the state the expression

```
 \working_space(myClass.m(\result))
```

is evaluated in as we only know that `\result!=0` holds in this state. Thus this expression equals the maximum of the specified working space clauses of both specification cases, which is `8`.

As example (31) illustrated, an expression `\working_space(m())` used in the working space clause of a specification case of a method `m2` denotes the worst case memory consumption of `m` as derivable from those of `m`'s specification cases, whose precondition can potentially be true[1] in the state `\working_space(m())` is evaluated in (either

[1] This means, in other words, it is not constradictory to the pre condition (if the `\working_space(m())` is evaluated in the pre state of `m2`) of the specification case

the pre or post state of `m2`, depending on whether the expression `\working_space(m())` occurs inside an `\old` expression or not). One could argue that a more fine grained `\working_space` function permitting to refer to the working space of a single specification case, would be desirable in order to be able to write more precise working space specifications.

Besides the fact that these restrictions can be considered inconvenient, they could easily give rise to specification bugs as the following example shows:

```
static SomeClass instance;

/*@ working_space 0; @*/
public static clear(){ instance = null; }

/*@ public normal_behavior
  @  requires instance == null;
  @  working_space \space(new SomeClass());
  @ also public normal_behavior
  @  requires instance != null;
  @  working_space 0;
  @*/
public static instance(){
    if(instance==null) instance = new SomeClass();
    return instance;
}

/*@  requires instance!=null;
  @  assignable instance;
  @  ensures \old(instance) != instance;
  @  working_space \working_space(clear()) +
  @                \working_space(instance());
  @*/
public SomeClass freshInstance(){
    clear();
    return instance();
}
```

The above specification of `freshInstance()` is incorrect since in the state `instance()` is called `instance==`**null** holds while in the poststate which is the state the expression `\working_space(instance())` is evaluated in `instance!=`**null** holds leading to a specified working space of `0` for this case. Writing `\old(\working_space(instance()))` does not help either since in the prestate `instance` is also required not to be **null**. This example illustrates that (i) it is not possible with the JML semantics as it is to specify the working space of `freshInstance()` relative to the working space of `getInstance()` and `clear()` since one does not have access to the state in which `getInstance()` is called in the code and (ii) that the semantics of the `\working_space` expression makes JML

`\working_space(m())` occurs in or to its post condition (if `\working_space(m())` is evaluated in the post state) respectively.

vulnerable to specification bugs since the programmer could be tempted to use method calls occurring in the specified method's body by just copying and pasting them into the `working_space` clause as has happened in the above specification.

3.2 JML Heap Memory Specifications in KeY

This work we propose propose an extension and modification of JML's `\working_space` function addressing both drawbacks of the current concept: the restriction to the pre and post state of the specified method and the lack of granularity. To distinguish the present JML specification from our proposal we will refer to the latter as the KeYJML spec.

The syntax and semantics of KeYJML memory specifications differs in certain respects from the original JML specification. Concretely these differences concern (i) the applied integer semantics, (ii) the state in which working space clauses are interpreted, (iii) the arguments allowed for `\space` expressions, (iv) a modification of the `\working_space` function and (v) an extension of JML's loop specifications.

Since KeY supports mathematical (unbounded) integers, `\space` and `\working_space` expressions as well as the expressions contained in `working_space` clauses do not have the Java type **long** but are treated by KeY as mathematical integers. This makes verification tasks a bit easier since modulo arithmetics is no longer required for evaluating working space clauses themselves[2] and prevents subtle (specification) bugs that can arise if integer overflows are not taken into account.

In contrast to JML, `working_space` clauses in KeYJML are interpreted in the prestate. This is motivated by the rationale that in scenarios in which performance and especially memory consumption specifications are of interest, namely for real-time and embedded systems with a possibly very limited amount of physical memory or scoped memory in a certain scope (as in RTSJ), one important application of `working_space` clauses could be to decide in the design or implementation phase whether a method can be called at a certain point of the program. Given this scenario, it makes sense that one should be able to evaluate the working space expressions based on the information that is available at the point the method is called which is its pre state. However, the approach presented in this work does not depend on this decision and keeping JML's original semantics would only require minor modifications in some of the calculus rules shown in section 5.

We also restrict the expressions allowed in `\space` expressions to (object as well as array) constructor calls. In the case of object constructors, this enables us to compute concrete values for `\space` expressions which would not be possible for arbitrary expressions of which we do not necessarily know their runtime type. When compiling JML specifications to their JAVA CARD DL counterparts `\space` expressions can then be directly replaced by concrete values.

The `\working_space` function undergoes the most distinct changes syntactically and semantically compared to the original JML definition. Beside supporting the original JML working space function KeYJML incorporates a new working space function that is written as `\working_space(m, pre)`, where `m` is a method signature and `pre` is a **boolean** expression. `\working_space(m, pre)` then denotes the maximum of specified

[2] Of course we still need modulo arithmetics for reasoning over programs containing arithmetic operations on integers.

amounts of heap space consumed by `m` if invoked in a state satisfying `pre`. This means that we have to take the maximum over all working space clauses having a precondition which is not contradictory to `pre`. If `pre` is chosen carefully this is only the case for exactly one contract. The second argument `pre` can be thought of as quoted meaning it is not evaluated in the state the containing expression `\working_space(m, pre)` occurs in, thus, making the entire expression `\working_space(m, pre)` not state dependent (we call an expression whose evaluation is not state dependent rigid). The expression

```
\working_space(SomeClass.m(int a1, int a2), a1<a2)
```

for instance denotes the maximum amount of heap space method `SomeClass.m` can consume according to its specification if invoked in a state in which the first of `m`'s arguments is smaller than the second one (`a1<a2`). This alternative suggestion of defining the `\working_space` function helps to overcome the drawbacks described in section 3.1 since (i) by making the prestate (of the method) and thus the relevant specifications explicitly selectable, it supports a finer level of granularity than the original JML variant of the `\working_space` function and (ii) reduces the risk of "copy and paste" related specification bugs as demonstrated in section 3.1.

Since JML lacks features for specifying the memory consumption of loops we propose a new loop specification clause we call `working_space_single_iteration` (`wssi`) which specifies the maximum amount of heap memory used by any single loop iteration not terminating with an exception or by a **break** statement (**continue** is allowed however). An upper bound of the accumulated amount of memory consumed by the loop in all its normally terminating iterations is then given by $dec * w$, where *dec* and *w* are the prestate values of the expressions specified by the `decreasing` and the `wssi` clause. The `decreasing` clause specifies a value that is (i) strictly decreasing in every iteration of the loop and (ii) always greater 0. Thus *dec* constitutes an upper bound for the number of loop iterations. The specification of `initArr` shown below illustrates the usage of the `wssi` clause:

```
/*@ public behavior
  @  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();
     }
 }
```

In each normally terminating iteration of the above **while** loop an object of type `Object` is created. Thus, `\space(`**new** `Object())` is a correct `wssi` clause. In case the runtime type of `a` is a strict subtype of `Object[]`, an `ArrayStoreException` is raised and the memory consumption of the loop body would be the working space of the constructor call **new** `ArrayStoreException()` which includes the space occupied by the newly created `ArrayStoreException` itself. Since , if this happens, the loop body doesn't terminate normally, this case need not be taken into account according to our definition of the semantics of wssi. However, it has to be taken into account when specifying `initArr`'s working space. This example also clarifies the rationale behind defining the `wssi` clause only for normally terminating iterations of the loop: The working space of **new** `ArrayStoreException()` (several hundreds of bytes, depending on the created stack trace) is significantly larger then the space occupied by a newly created object of type `Object` (8 bytes for the JVM characteristics we use). If w were also required to be an upper bound for the heap space consumed by an abruptly terminating iteration the value $dec * w$ would be of no real significance for the worst case memory consumption estimation of the loop, since:

- If the loop raises no exception w is by an order of magnitudes larger then the space actually consumed by each iteration (`\space(`**new** `Object())`) which also applies to the worst case estimation $dec * w$ for the memory consumption of the entire loop.
- If the loop raises an exception, it is only executed once and our worst case estimation $dec * w$ is wrong by factor dec.

By restricting `wssi` the way it is done we get at least a more precise worst case estimation for the first of the above two cases. This information can also be used by the KeY tool for determining a correct upper bound for the memory consumption of a loop terminating abruptly in an arbitrary iteration as shown in section 5.

Assumptions on the Java Virtual Machine The approach presented in this work is independent of characteristics, such as the memory overhead needed to store an object and alignment issues, of the Java Virtual Machine (JVM) the regarded code runs on. Nevertheless for the calculus rules and examples presented in the following we will assume concrete JVM characteristics, namely the ones observable for the Sun J2SE 1.4.2 VM running on the Linux operating system. This entails that we can provide concrete values for the space occupied by objects and arrays (see section 5), in case their dimension is known, which should make rules and examples appearing in the following more illustrative.

In particular the JVM characteristics we assume are:

- The space $as_{e,l}$ required for a one-dimensional array of length l with each of its entries occupying e bytes is:

$$as_{e,l} = min\{a | a \geq 12 + e * l \wedge a \; mod \; 8 \equiv 0\}$$

- The space $space_T$ in bytes occupied by an object of type T is:

$$space_T := min\{a | a \geq 8 + s \wedge a \; mod \; 8 \equiv 0\}$$

  where s is the space occupied by the fields of the object.

Mapping JML expressions to JAVA CARD DL In order to make JML expressions utilizable within KeY, it is necessary to compile them to JAVA CARD DL.

Let $\mathcal{T}$ be a mapping from JML expressions to JAVA CARD DL terms and formulas. For the JML functions `\working_space` and `\space` we define $\mathcal{T}$ as:

- $\mathcal{T}(\texttt{\textbackslash working\_space(m, pre)}) := ws^r_{m,pre}$,
  with $ws^r_{m,pre}$ being a rigid term meaning its value is not state dependent. This means in particular that *pre* can be thought of as quoted since it is not evaluated in the state $ws^r_{m,pre}$ occurs in.
- $\mathcal{T}(\texttt{\textbackslash working\_space(m(a1,...,an))}) := ws^{nr}_{m(a_1,\ldots,a_n)}$,
  with $ws^{nr}_{m(a_1,\ldots,a_n)}$ being a non-rigid term.
- $\mathcal{T}(\texttt{\textbackslash space(}\textbf{new}\ \texttt{T())}) := i$,
  with the integer literal i being the amount of heap space an object of type T occupies.
- For representing `\space` expressions whose argument has an array type we introduce a new rigid function symbol $space^{arr}$, where $space^{arr}(s,l)$ denotes the space occupied by a one-dimensional array of length l whose entries (for primitive typed entries) or entry references (in case of reference typed entries) respectively have size s.
  $\mathcal{T}(\texttt{\textbackslash space(}\textbf{new}\ \texttt{T[d1]...[dn][]...[])}) :=$
  $space^{arr}(4,d1) +$
  $d1 * \mathcal{T}(\texttt{\textbackslash space(}\textbf{new}\ \texttt{T[d2]...[dn][]...[])})$,
  $\mathcal{T}(\texttt{\textbackslash space(}\textbf{new}\ \texttt{T[d][]...[])}) := space^{arr}(4,d)$
  and $\mathcal{T}(\texttt{\textbackslash space(}\textbf{new}\ \texttt{T[d])}) := space^{arr}(s,d)$,
  where
  $$s := \begin{cases} 1 & iff\ T \in \{byte, boolean\} \\ 2 & iff\ T \in \{short, char\} \\ 4 & iff\ T = int\ or\ T\ is\ a\ reference\ type \\ 8 & iff\ T = long \end{cases}$$
  Although $space^{arr}$ is a rigid function, terms having $space^{arr}$ as top level function symbol can be non-rigid since the second argument of a $space^{arr}$ term can be non-rigid.

Example 32 (Heap Space Terms) *The formula*

$$\{v := 2\} space^{arr}(4,v)$$

is logically equivalent to the formula

$$space^{arr}(4,2)$$

since $space^{arr}(4,v)$ *is non-rigid. In contrast*

$$\{v := 2\} ws^r_{m,v\geq 0}$$

is equivalent to

$$ws^r_{m,v\geq 0}$$

since $ws^r_{m,v\geq 0}$ *is rigid and the state update does not affect the formula* $v \geq 0$ *in the index.*

4 Proof Obligations

For checking the correctness of JML method contracts these contracts are compiled to JAVA CARD DL formulas that are valid iff the contract they are based is correct. The correctness of these formulas, which we call proof obligations (POs), can be checked by the KeY system.

For reasoning over memory performance aspects of JML contracts in JAVA CARD DL, we introduce a program variable **h** representing the current amount of used heap space. This variable is increased appropriately whenever a statement consuming memory is symbolically executed. Since the `working_space` clause defines an upper bound for the amount of heap memory consumed by the specified method, the corresponding PO must state that the value of **h** is increased (relative to the prestate) by at most this amount. Thus, for instance, the PO expressing the validity of the performance contract for method `doSth` shown on page 3.1 has the form:

$$PRE \rightarrow \{\mathbf{h}_{max} := \mathbf{h} + S\}\langle \texttt{doSth();} \rangle \mathbf{h} \leq \mathbf{h}_{max} \tag{3}$$

where S is the working space of `doSth` as specified in the considered performance contract (like other JML expressions occurring the contract working space expression is translated to JAVA CARD DL). Since we decided to evaluate the working space in the prestate of `doSth`, we introduced a fresh program variable $\mathbf{h}_{max}$ for storing the prestate value of $\mathbf{h} + S$ which is then compared to **h** in the poststate ($\mathbf{h} \leq \mathbf{h}_{max}$).

Remark 41 (JML and KeYJML) *If we apply the original JML semantics, namely that the working space clause is evaluated in the post state the resulting PO changes only slightly:*

$$PRE \rightarrow \{\mathbf{h}_{old} := \mathbf{h}\}\langle \texttt{doSth();} \rangle \mathbf{h} \leq \mathbf{h}_{old} + S \tag{4}$$

5 Calculus Rules

We now turn to calculus rules extending the existing JAVA CARD DL calculus provided by KeY. These newly defined rules reflect the semantics of KeYJML and JAVA CARD DL expressions described in section 3.2 and make JAVA CARD DL suitable for reasoning over memory performance aspects of JAVA CARD programs.

Remark 51 *We write $\Gamma \Rightarrow \Delta \ni \varphi$ iff either $\varphi \in \Gamma \cup \Delta$ or there is a formula Φ such that $\Phi \in \Gamma \cup \Delta$ and φ is subformula of Φ. Analogously we define that $\Gamma \Rightarrow \Delta \ni t$ holds iff there is a formula Φ such that $\Phi \in \Gamma \cup \Delta$ and t is subterm of Φ.*

For two working space terms ws^r_{m,φ_1} and ws^r_{m,φ_2} with $\varphi_1 \rightarrow \varphi_2$ the maximum amount of heap space consumed by m under the precondition φ_1 cannot be larger then under the precondition φ_2 since the set of states satisfying φ_1 is a subset of the set of states satisfying φ_2 (or $\{s | s \models \varphi_1\} \subseteq \{s | s \models \varphi_2\}$). This leads us to the rule:

$$\text{wsRigid}\ \frac{\Gamma \Rightarrow \{*\}(\varphi_1 \rightarrow \varphi_2),\ \Delta \quad ws^r_{m,\varphi_1} \leq ws^r_{m,\varphi_2},\ \Gamma \Rightarrow \Delta}{\Gamma \Rightarrow \Delta \ni ws^r_{m,\varphi_1},\ ws^r_{m,\varphi_2}}$$

where $\{*\}$ is an anonymous update assigning a fresh constant to every location in $\varphi_1 \rightarrow \varphi_2$ and thus erasing all context information from Γ and Δ that could influence the validity of $\varphi_1 \rightarrow \varphi_2$. The reason for applying $\{*\}$ in the first premise is that we need to show that that $\varphi_1 \rightarrow \varphi_2$ holds in an arbitrary state (and not only in the states satisfying $\Gamma \Rightarrow \Delta$). In the second premise we can then use $ws^r_{m,\varphi_1} \leq ws^r_{m,\varphi_2}$ as an assumption (meaning it becomes part of the antecedent).

We know that a method can have no negative memory consumption which is reflected in the rules *wsGEqZeroR* and *wsGEqZeroNR*:

$$\mathsf{wsGEqZeroR}\ \frac{\Gamma, ws^r_{m,\varphi} \geq 0\ \Rightarrow \Delta}{\Gamma \Rightarrow \Delta \ni ws^r_{m,\varphi}}$$

$$\mathsf{wsGEqZeroNR}\ \frac{\Gamma, \{\mathcal{U}\} ws^{nr}_{m(a_1,\ldots,a_n)} \geq 0\ \Rightarrow \Delta}{\Gamma \Rightarrow \Delta \ni \{\mathcal{U}\} ws^{nr}_{m(a_1,\ldots,a_n)}}$$

where $\{\mathcal{U}\}$ is an arbitrary state update. If the length of an array is known (meaning it is a concrete value not only a symbolic expression) the heap space consumed by this array can be determined:

$$\mathsf{arraySpaceConcreteDim}\ \frac{as_{e,l}}{space^{arr}(e,l)}$$

where

- e and l are integer literals.
- $as_{e,l} := min\{a | a \geq 12 + e * l \wedge a\ mod\ 8 \equiv 0\}$

Above we defined $as_{e,l}$ in accordance with the characteristics of the Sun JVM implementation running on Linux (see section 3.2) which means that the space (measured in bytes) occupied by an array is a multiple of 8 and the overhead (the space occupied by an array without being available to store array elements) of an array is 12 bytes.

In case the length of an array is not known, we can still determine upper and lower bounds of the $space^{arr}(e,l)$ term depending on the value of l:

$$\mathsf{arraySizeLowerUpperBound}\ \frac{\begin{array}{l} space^{arr}(e,l) \leq ub(e,l) \wedge \\ space^{arr}(e,l) \geq lb(e,l) \wedge \\ space^{arr}(e,l) \geq min_{as},\ \Gamma \Rightarrow \Delta \end{array}}{\Gamma \Rightarrow \Delta \ni space^{arr}(e,l)}$$

where

- $ub(e,l)$ denotes the least upper bound of $space^{arr}(e,l)$ for arbitrary values of l. For the VM implementation we consider we get for instance:

$$ub(e,l) := \begin{cases} 8l + 16, & \text{if } e = 8 \\ e(l-1) + 20, & \text{if } e \in \{1,2,4\} \end{cases}$$

– $lb(e,l)$ denotes the greatest lower bound of $space^{arr}(e,l)$ for arbitrary values of l:

$$lb(e,l) := \begin{cases} 8l+16, & \text{if } e = 8 \\ el+12, & \text{if } e \in \{1,2,4\} \end{cases}$$

– $min_{as} := space^{arr}(e,0)$ is the size of an array of length 0 which is for the VM we consider 16 bytes.

The symbolic execution of a constructor call increases **h** by the size of the created object:

$$\text{objectCreation}\ \frac{\Gamma \Rightarrow \{\mathcal{U}; \mathbf{h} := \mathbf{h} + space_T\}\langle \pi\, \texttt{OC}\, \omega\rangle \phi,\ \Delta}{\Gamma \Rightarrow \{\mathcal{U}\}\langle \pi\, \texttt{v=}\mathbf{new}\ \texttt{T(a1,...,an);}\ \omega\rangle \phi,\ \Delta}$$

– $space_T$ is an integer literal representing the heap space occupied by an object of dynamic type T which is in this case (see section 3.2)

$$space_T := min\{a | a \geq 8 + s \wedge a\ mod\ 8 \equiv 0\}$$

where s is the space occupied by the fields of the object (for a non-primitive field only the space occupied by the reference, namely 4 bytes, not the object itself).

– `OC` stands for the code modeling the object's creation and initialization and the execution of the constructor `T(a1,...,an)`.

Array constructors are treated in a similar way with the mere difference that the size of an array cannot necessarily be statically determined since it depends on the array's dimension. For the sake of readability we only consider the case of a one-dimensional array here:

$$\text{arrayCreation}\ \frac{\Gamma \Rightarrow \{\mathcal{U}; \mathbf{h} := \mathbf{h} + space^{arr}(e, \texttt{d1})\}\langle \pi\, \texttt{AC}\, \omega\rangle \phi,\ \Delta}{\Gamma \Rightarrow \{\mathcal{U}\}\langle \pi\, \texttt{v=}\mathbf{new}\ \texttt{T[d1]);}\ \omega\rangle \phi,\ \Delta}$$

Where `AC` is a placeholder for the code modeling the array's creation.

In order to be also able to modularly verify performance contracts (as also proposed in [12]), we need a rule describing the effect of a method's execution merely by utilizing the information retrievable from its specification instead of symbolically executing the method body. For the sake of simplicity we now consider only `normal_behavior` method contracts that allow only normal termination (without an exception) of the specified method leading to a more compact rule. Let C be a `normal_behavior` contract for a method `m()` derived from a JML specification and let the JAVA CARD DL formulas *Pre* and *Post* be the pre- and postcondition as defined by C, *Mod* the set of locations that are permitted to be modified by `m()` (obtained from the `assignable` clause) and w the term obtained from C's `working_space` clause. Let further be $\mathcal{V}$ an update assigning fresh constants to all locations in *Mod* as a means of reflecting the state change caused by `m()`. For the sake of readability we only consider the contract rule for a parameterless static method here:

$$\text{applyContract}\ \frac{\begin{array}{l}\Gamma \Rightarrow \{\mathcal{U}\} Pre,\ \Delta \\ \Gamma \Rightarrow \{\mathcal{U}\}(ws^{nr}_{\texttt{m()}} = w \rightarrow \\ \{\mathcal{V} \| \mathbf{h} := \mathbf{h} + ws^{nr}_{\texttt{m()}}\}(Post \rightarrow \langle \pi\, \omega\rangle \phi)),\ \Delta\end{array}}{\Gamma \Rightarrow \{\mathcal{U}\}\langle \pi\, \texttt{m();}\ \omega\rangle \phi,\ \Delta}$$

The first premise of rule applyContract states that the precondition *Pre* is required to hold in the state in which `m()` is invoked. In the second premise we can then use the information that after the execution of `m()` the postcondition *Post* holds. As we can see applyContract also describes that in every state s reachable by state update $\mathcal{U}$ the worst case memory consumption of $m()$ (when executed in state s) equals w evaluated in state s (indicated by $ws^{nr}_{\mathtt{m()}} = w$ and the state update $\mathbf{h} := \mathbf{h} + w$).

Remark 52 (Working space terms in applyContract**)** *The usage of $ws^{nr}_{m()}$ could seem to be counter intuitive on a first glance since the rule*

$$\frac{\Gamma \Rightarrow \{\mathcal{U}\}Pre,\ \Delta \quad \Gamma \Rightarrow \{\mathcal{U}\}\{\mathcal{V}||\mathbf{h} := \mathbf{h} + w\}(Post \rightarrow \langle \pi\, \omega \rangle \phi),\ \Delta}{\Gamma \Rightarrow \{\mathcal{U}\}\langle \pi\, m();\ \omega \rangle \phi,\ \Delta}$$

also expresses that $\mathbf{h}$ *is increased by the value w specified by the applied contract.*

However, keeping the information explicitely in the sequent that w equals the memory consumption of m when called in the state defined by $\mathcal{U}$ and the sequent context Γ and Δ (as expressed by the subformula $ws^{nr}_{m()} = w$ in rule applyContract*) can ease proving tasks later on in case rigid working space terms occur in ϕ. This is mainly owed to the rule* wsNonRigid *allowing to directly relate rigid and non-rigid working space terms refering to the same method.*

Remark 53 (JML and KeYJML) *Basing the rule set on the original JML semantics that demands evaluation of the working space clause in the post state would result in the following contract rule:*

$$\mathsf{applyContract}' \ \frac{\begin{array}{l}\Gamma \Rightarrow \{\mathcal{U}\}Pre,\ \Delta \\ \Gamma \Rightarrow \{\mathcal{U};\mathcal{V};(ws^{nr}_{m()} = w \rightarrow \\ \{\mathbf{h} := \mathbf{h} + ws^{nr}_{m()}\}(Post \rightarrow \langle \pi\, \omega \rangle \phi)),\ \Delta\end{array}}{\Gamma \Rightarrow \{\mathcal{U}\}\langle \pi\, m();\ \omega \rangle \phi,\ \Delta}$$

The relation between a non-rigid working space term and a rigid one can be defined in similar manner as done by rule wsRigid for two rigid working space terms:

$$\mathsf{wsNonRigid} \ \frac{\Gamma \Rightarrow \{\mathcal{U}\}\varphi,\ \Delta \quad \Gamma,\ \{\mathcal{U}\}ws^{nr}_{m} \leq ws^{r}_{m,\varphi} \Rightarrow \Delta}{\Gamma \Rightarrow \Delta \ni \{\mathcal{U}\}ws^{nr}_{m(a_1,\ldots,a_n)}, ws^{r}_{m,\varphi}}$$

If the precondition φ is valid in the symbolic state s the nonrigid working space term $ws^{nr}_{m(a_1,\ldots,a_n)}$ occurs in (first premise) the set S_1 of concrete program states defined by s is a subset of the set S_2 of conrete states which are a model of φ. Thus according to the semantics of ws^{nr} and ws^{r} the relation $\{\mathcal{U}\}ws^{nr}_{m(a_1,\ldots,a_n)} \leq ws^{r}_{m,\varphi}$ holds (second premise).

With the rules defined so far, it is not yet possible to put a working space term $ws^{r}_{m,\varphi}$ in relation to the working spaces specified by any of m's contracts. However, we know that if there is a contract C for m whose precondition is logically weaker than φ, the semantics of $ws^{r}_{m,\varphi}$ entails that $ws^{r}_{m,\varphi} = t$, with t being the specified working space

of C, holds in states satisfying φ. Analogously, we can define a rule for the case that φ is weaker than C's precondition in which every value t, when evaluated in a state satisfying *Pre*, is a lower bound for $ws^r_{m,\varphi}$. This results in two calculus rules for the two mentioned cases:

$$\mathsf{wsContract1}\ \frac{\Gamma \Rightarrow \{*\}(\varphi \rightarrow Pre),\ \Delta \quad \Gamma,\ \{*\}(\varphi \wedge ws^r_{m,\varphi} = t) \Rightarrow \Delta}{\Gamma \Rightarrow \Delta \ni ws^r_{m,\varphi}}$$

$$\mathsf{wsContract2}\ \frac{\Gamma \Rightarrow \{*\}(Pre \rightarrow \varphi),\ \Delta \quad \Gamma,\ \{*\}(Pre \wedge t \leq ws^r_{m,\varphi}) \Rightarrow \Delta}{\Gamma \Rightarrow \Delta \ni ws^r_{m,\varphi}}$$

The motivation for the anonymizing updates $\{*\}$ used in each of the above rule's first premises is the same as for rule rigidWS namely that, for instance, the implication $Pre \rightarrow \varphi$ (as occurring in rule wsContract2) has to be valid, meaning it is required to hold in every state not only the ones determined by the context formulas Γ and Δ. Since the working space t is only defined for states meeting *Pre*, all we can assume in each of the second premises is that, for instance, the relation $t \leq ws^r_{m,\varphi}$ holds in a state satisfying *Pre*. This consideration leads us to the formula $\{*\}(Pre \wedge t \leq ws^r_{m,\varphi})$ that is part of the antecedent of the second premise of rule wsContract2.

For non-rigid working space terms we can define a similar rule motivated basically by the same considerations as wsContract1 with t and *Pre* as defined above:

$$\mathsf{wsContract3}\ \frac{\Gamma \Rightarrow \{\mathcal{U}\}Pre,\ \Delta \quad \Gamma,\ \{\mathcal{U}\}(ws^{nr}_{m(a_1,\ldots,a_n)} = \{\mathcal{V}\}t) \Rightarrow \Delta}{\Gamma \Rightarrow \Delta \ni ws^{nr}_{m(a_1,\ldots,a_n)}}$$

We use the update $\{\mathcal{V}\} := \{p_1 := a_1 || \ldots || p_n := a_n\}$ to map m's parameters $p_1, \ldots, p_n$ occuring in t to the concrete arguments $a_1, \ldots, a_n$ taken from $ws^{nr}_{m(a_1,\ldots,a_n)}$.

Remark 54 (Soundness of Contract Rules) *For the rules wsContract1 and wsContract3 to be sound we have to require that for any pair C_1, C_2 of specification cases for a method m the condition $\phi_1 \wedge \phi_2 \rightarrow w_1 = w_2$ holds, where ϕ_i and w_i denote the precondition and working space of contract C_i. For this condition to be true it is sufficient to require that different specification cases for the same method have disjoint preconditions.*

In case $\phi_1 \wedge \phi_2 \rightarrow w_1 = w_2$ does not hold, as for instance if we set $\phi_1 := \phi_2 :=$ true and $w_1 := 0$, $w_2 := 1$, using wsContract3 *we could for instance prove that the unsatisfiable (according to the semantics of ws^{nr}) formula $ws^{nr}_{m()} < 0$ holds as these derivation steps illustrate:*

$$\frac{\dfrac{*}{\Rightarrow true, ws^{nr}_{m()} < 0} \quad ws^{nr}_{m()} = 0 \Rightarrow ws^r_m < 0}{\Rightarrow ws^{nr}_{m()} < 0}$$

By applying wsContract3 *again to the remaining goal $ws^{nr}_{m()} = 0 \Rightarrow ws^r_m < 0$ using the second contract for m we get a proof tree with one open goal of the form*

$$ws^{nr}_{m()} = 0, ws^{nr}_{m()} = 1 \Rightarrow ws^r_m < 0$$

which can eventually also be closed:

$$\dfrac{\dfrac{\dfrac{*}{ws^{nr}_{m()} = 0, ws^{nr}_{m()} = 1, false \Rightarrow ws^{r}_{m} < 0}}{ws^{nr}_{m()} = 0, ws^{nr}_{m()} = 1, 0 = 1 \Rightarrow ws^{r}_{m} < 0}}{ws^{nr}_{m()} = 0, ws^{nr}_{m()} = 1 \Rightarrow ws^{r}_{m} < 0}$$

Remark 55 (JML and KeYJML) *Applying the JML semantics leads to the following working space contract rules:*

$$\text{wsContract1}' \ \dfrac{\Gamma \Rightarrow \{*\}(\varphi \rightarrow Pre), \Delta \quad \Gamma, \{*\}(\varphi \wedge \mathcal{V}(Mod)(Post \wedge ws^{r}_{m,\varphi} = t)) \Rightarrow \Delta}{\Gamma \Rightarrow \Delta \ni ws^{r}_{m,\varphi}}$$

$$\text{wsContract2}' \ \dfrac{\Gamma \Rightarrow \{*\}(Pre \rightarrow \varphi), \Delta \quad \Gamma, \{*\}(Post \wedge t \leq ws^{r}_{m,\varphi}) \Rightarrow \Delta}{\Gamma \Rightarrow \Delta \ni ws^{r}_{m,\varphi}}$$

$$\text{wsContract3}' \ \dfrac{\Gamma \Rightarrow \{\mathcal{U}\}Pre, \Delta \quad \Gamma, \{\mathcal{U}\}(ws^{nr}_{m(a_1,\ldots,a_n)} = \{\mathcal{V}'\}t \wedge \{\mathcal{V}'\}Post) \Rightarrow \Delta}{\Gamma \Rightarrow \Delta \ni ws^{nr}_{m(a_1,\ldots,a_n)}}$$

Where Pre and Post are the pre and post condition of the applied method contract, t its working space and Mod its modifier set. In addition we define

$$\mathcal{V}' := \{\mathcal{V}; \mathcal{V}(Mod)\}$$

and

$$\mathcal{V} := \{p_1 := a_1 || \ldots || p_n := a_n\}$$

As the last step of adapting the JAVA CARD DL calculus to performance verification needs we will have a look at a loop invariant rule making use of loop annotations provided by a JML specification:

- a loop invariant *Inv* holding at the beginning of each and the end of each normally terminating (meaning not termination by **break** or an *exception*) loop iteration.
- an `assignable` clause defining a set of locations *Mod* modifiable by the loop. Allowing the specification of assignable clauses for loops is a KeY-specific extension of JML.
- a `decreasing` clause providing a term *var* that is strictly decreasing in each iteration of the loop while remaining greater or equal to 0 thus inducing termination of the loop.
- a working space *wsl* for a single normally terminating iteration of the loop obtained from the, again KeY-specific, `working_space_single_iteration` clause.

$$\textsf{loopInvTotal}\ \frac{\begin{array}{l}\Gamma \Rightarrow \{\mathcal{U}\}(Inv \wedge var \geq 0),\ \Delta \\ \Gamma,\ \{\mathcal{U}; \mathcal{V}_1(Mod)\}(Inv \wedge se) \Rightarrow \\ \quad \{\mathcal{U}; i := \mathbf{h}; j := wsl; \\ \quad\quad k := var; \mathcal{V}_1(Mod)\}[\texttt{tc(p,e)}]\psi_1,\ \Delta \\ \Gamma,\ \{\mathcal{U}; \mathcal{V}_2(Mod)\}(Inv \wedge se \wedge var \geq 0) \Rightarrow \\ \quad \{\mathcal{U}; \mathcal{V}_2(Mod); k := var\}\langle\texttt{tc(p,e)}\rangle\psi_2,\ \Delta \\ \Gamma,\ \{\mathcal{U}; \mathcal{V}_3(Mod)\}(Inv \wedge \neg se) \Rightarrow \\ \quad \{\mathcal{U}; \mathbf{h} := \mathbf{h} + var * wsl; \mathcal{V}_3(Mod)\}\langle\pi\,\omega\rangle\phi,\ \Delta\end{array}}{\Gamma \Rightarrow \{\mathcal{U}\}\langle\pi\,\textbf{while}\,\texttt{(se)\{p\}}\,\omega\rangle\phi,\ \Delta}$$

The first premise of loopInvTotal states that the invariant *Inv* is valid just before the loop is executed the first time.

The second premise states that, in case the loop body terminates normally, the loop invariant is preserved by the loop body and the loop guard[3] and the memory consumption of this loop iteration does not exceed *wsl*. Here

- $\mathcal{V}_1(Mod)$ is a parallel update assigning fresh constants to the locations in *Mod*.
- `tc(p,e)` is derived from the original loop body that was transformed in a way that, for instance, allows capturing the execution of **break** statements or the raising of an uncaught (i.e. uncaught within the loop body) exception. Such events are memorized by fresh program variables.
- $\psi_1 := \psi_{exc_1} \wedge \psi_{break_1} \wedge \psi_{normal_1}$
  ψ_1 describes the conditions required to hold after the loop body terminated normally (ψ_{normal_1}) or abruptly by an exception (ψ_{exc_1}) or break (ψ_{break_1}).
  - $\psi_{normal_1} := (e = \texttt{null} \wedge b_{break} = FALSE \vee b_{cont} = TRUE) \rightarrow (inv \wedge \mathbf{h} - i \leq j)$
    The rationale behind this formula is that if no exception has been thrown and the loop body terminates normally (indicated by the values of e and b_{break_1}) or a continue statement has been executed (indicated by $b_{cont} = TRUE$) then the invariant holds and **h** was increased at most by the pre state value of *wsl*.
  - $\psi_{exc_1} := e \neq \texttt{null} \rightarrow \{\mathbf{h} := \mathbf{h} + j * (k - var - 1)\}[\pi\,\textbf{throw}\ \texttt{e;}\ \omega]\phi$
    If an exception has been thrown the value of **h** is increased by $j * (k - var - 1)$ which is the maximal cumulated amount of heap space the loop has potentially consumed in all (normally terminating) iterations executed before the observed iteration leading to the exception has been reached. The heap space consumed by the last iteration (the one raising the exception) was already added to **h** during the symbolic execution of the loop body. In addition after the execution of the rest of the program π **throw** `e;` ω the postcondition ϕ must hold. We inject the statement **throw** `e;` into the remaining code since the exception was not caught in the original implementation of the loop but only by a **catch** block added by the transformation `tc` for memorizing uncaught exceptions.
  - $\psi_{break_1} := b_{break} = true \rightarrow \{\mathbf{h} := \mathbf{h} + j * (k - var - 1)\}[\pi\,\omega]\phi$
    In case a **break** statement terminates the loop we are in a similar situation as in the exceptional case to that effect that **h** is also increased by $j * (k - var - 1)$

[3] The loop guard can potentially have side effects.

and that the postcondition ϕ must be shown to hold after the remaining code is executed.

Recapitulating one can say that these explanations about ψ_1 also clarify why it is sufficient to specify only the memory consumption of normally terminating loop iterations: All information needed about abruptly terminating iterations is obtained by symbolic execution and thus does not need to be part of the specification.

The third premise covers the termination function obtained from the `decreasing` clause by stating that *var* has to decrease strictly if the loop body terminates normally. Again we have to distinguish between the normal and the two kinds of abrupt termination:

$$\psi_2 := \psi_{exc_2} \wedge \psi_{break_2} \wedge \psi_{normal_2}$$

where

- $(e = \texttt{null} \wedge b_{break} = true \vee b_{cont} = TRUE) \rightarrow (var \geq 0 \wedge var < k)$: In case of non-abrupt termination of the loop body (and guard) $var \geq 0 \wedge var < k$ holds with k being the prestate value of *var*.
- $\psi_{exc_2} := e \neq \texttt{null} \rightarrow \langle \pi \textbf{ throw } \texttt{e;}\ \omega \rangle\, true$: If an exception was thrown the remaining part π **throw** `e;` ω of the program must terminate.
- $\psi_{break_2} := b_{break} = true \rightarrow \langle \pi\, \omega \rangle\, true$: If a break occurs the remaining program $\pi\omega$ must terminate.

Finally in the fourth premise of rule loopInvTotal we can use the information that the loop invariant holds after the last iteration of the loop (at the end of which $\neg se$ holds) and that **h** was increased at this point by the prestate value of $var * wsl$, which is encoded in an update.

6 Examples

In the following we demonstrate the capabilities of the presented approach based on two examples. In the first one we examine modular verification of performance constraints. The second one shows the JML specification of a realistic piece of code with performance specifications that can be verified by KeY requiring only little user interaction.

6.1 Modular vs. Non-Modular Verification

We will now regard the modular verification of performance contracts illustrated by a short example containing two static methods `create` and `init`. The method `create` creates an integer array of length `a` if the value of `a` that is passed as an argument is greater zero. `init` initializes the static field `arr` with an integer array of length 7 that was created by the `create` method. This behavior and the corresponding heap space consumption is specified by JML method contracts.

```
private static int[] arr;
...
/*@ public normal_behavior
```

```
  @  requires a>0;
  @  assignable \nothing;
  @  working_space 16+a*4;
  @ also public normal_behavior
  @  requires a<=0;
  @  assignable \nothing;
  @  working_space 0;
  @*/
public static int[] create(int a){
    if(a>0){
        return new int[a];
    }
    return null;
}

/*@ public normal_behavior
  @  assignable arr;
  @  working_space
  @    \working_space(create(int arg),arg==7);
  @*/
public static void init(){
    arr = create(7);
}
```

The proof obligations derived from the two contracts for `create` are

$$a > 0 \rightarrow \{\mathbf{h}_{max} := \mathbf{h} + 16 + a * 4\}\langle \texttt{create();} \rangle\, \mathbf{h} \leq \mathbf{h}_{max} \tag{5}$$

and

$$a \leq 0 \rightarrow \{\mathbf{h}_{max} := \mathbf{h}\}\langle \texttt{create();} \rangle\, \mathbf{h} \leq \mathbf{h}_{max} \tag{6}$$

respectively. Both can be proven automatically by the KeY-prover using the extension described in Section 5. Of the rules defined in this sections the only ones used for these proof obligations were arrayCreation for the symbolic execution of the array constructor call and arraySizeLowerUpperBound for deriving that the upper bound of the amount of memory consumed by the created array is smaller or equal to $16 + a * 4$.

The proof obligation for `init`'s specification is given by the formula:

$$\{\mathbf{h}_{max} := \mathbf{h} + ws^{r}_{\texttt{create(}\mathbf{int}\ \texttt{a)}}(\texttt{a} = 7)\}\langle \texttt{init();} \rangle\, \mathbf{h} \leq \mathbf{h}_{max}$$

Since `init()` contains the method call `create(7)` we either have to symbolically execute `create`'s method body or make use of `create`'s method contract by applying the rule applyContract with the first one of `create`'s contracts.

Non-Modular Verification When symbolically executing the method body we get a sequent

$$\Rightarrow \mathbf{h} + 40 \leq \mathbf{h} + ws^{r}_{\texttt{create(}\mathbf{int}\ \texttt{a)}}(\texttt{a} = 7)$$

after the symbolic execution has terminated which shows that **h** was increased by 40 bytes and we now have to prove that this is smaller than the maximum specified amount of space consumed by the method `create(int a)` when called under the precondition $\texttt{a} = 7$. We can achieve this by applying rule wsContract1 (using again the first of `create`'s contracts) to $ws^r_{\texttt{create(int a)}}(\texttt{a} = 7)$ resulting in two goals

$$\Rightarrow \{*\}(\texttt{a} = 7 \rightarrow \texttt{a} > 0),\ \mathbf{h} + 40 \leq \mathbf{h} + ws^r_{\texttt{create(int a)}}(\texttt{a} = 7)$$

which is obviously valid since $\{*\}(\texttt{a} = 7 \rightarrow \texttt{a} > 0)$ is valid, and

$$\begin{array}{l} \{*\}(\texttt{a} = 7 \wedge 16 + \texttt{a} * 4 = ws^r_{\texttt{create(int a)}}(\texttt{a} = 7)) \\ \Rightarrow \\ \mathbf{h} + 40 \leq \mathbf{h} + ws^r_{\texttt{create(int a)}}(\texttt{a} = 7) \end{array}$$

which can be simplified to

$$\begin{array}{l} \{*\}\texttt{a} = 7,\ 16 + 7 * 4 = ws^r_{\texttt{create(int a)}}(\texttt{a} = 7)) \\ \Rightarrow \\ \mathbf{h} + 40 \leq \mathbf{h} + ws^r_{\texttt{create(int a)}}(\texttt{a} = 7) \end{array}$$

and eventually (since $16 + 7 * 4 = 44$ holds) to

$$\{*\}\texttt{a} = 7,\ 44 = ws^r_{\texttt{create(int a)}}(\texttt{a} = 7) \Rightarrow \mathbf{h} + 40 \leq \mathbf{h} + 44$$

whose validity is also obvious due to $\mathbf{h} + 40 \leq \mathbf{h} + 44$ occurring in the succedent.

Modular Verification In the other case if we decide to verify our program in a modular way by utilizing `create`'s contract and applying applyContract the symbolic execution of `create` leads to **h** being increased by 44. Thus after the symbolic execution of `init` is completed we have a sequent:

$$\Rightarrow \mathbf{h} + 44 \leq \mathbf{h} + ws^r_{\texttt{create(int a)}}(\texttt{a} = 7)$$

which can be shown to be valid by either applying wsContract1 (as done in the first case) or wsNonRigid which results in the two sequents

$$\begin{array}{l} \{\texttt{a} := 7\}ws^{nr}_{\texttt{create(int)}} \leq 44 \\ \Rightarrow \\ \{\texttt{a} := 7\}\texttt{a} = 7, \\ \mathbf{h} + \{\texttt{a} := 7\}ws^{nr}_{\texttt{create(int)}} \leq \mathbf{h} + ws^r_{\texttt{create(int a)}}(\texttt{a} = 7) \end{array}$$

which is valid since $\{\texttt{a} := 7\}\texttt{a} = 7$ (occurring in the succedent) can be simplified to $7 = 7$ and eventually to *true*, and

$$\begin{array}{l} \{\texttt{a} := 7\}ws^{nr}_{\texttt{create(int)}} \leq 44, \\ \{\texttt{a} := 7\}ws^{nr}_{\texttt{create(int)}} \leq ws^r_{\texttt{create(int a)}}(\texttt{a} = 7) \\ \Rightarrow \\ \mathbf{h} + \{\texttt{a} := 7\}ws^{nr}_{\texttt{create(int)}} \leq \mathbf{h} + ws^r_{\texttt{create(int a)}}(\texttt{a} = 7) \end{array}$$

which is also valid since the formula

$$\mathbf{h} + \{\mathtt{a} := 7\} ws^{nr}_{\mathtt{create}(\mathbf{int})} \leq \mathbf{h} + ws^{r}_{\mathtt{create}(\mathbf{int}\ \mathtt{a})}(\mathtt{a} = 7)$$

in the succedent is logically entailed by

$$\{\mathtt{a} := 7\} ws^{nr}_{\mathtt{create}(\mathbf{int})} \leq ws^{r}_{\mathtt{create}(\mathbf{int}\ \mathtt{a})}(\mathtt{a} = 7)$$

occurring in the antecedent.

6.2 Javolution

Javolution [5] is a real-time Java library facilitating the development of real-time compliant Java applications. This is accomplished by, for instance, reducing the need for garbage collection by using memory contexts in which objects, that are no longer needed, are recycled and can be reused the next time an object of the corresponding class is needed in the containing context. Javolution also provides time-deterministic implementations of standard Java packages such as collection and map data structures. The example we now consider is taken from the Javolution class `FastMap` which implements the same functionality as `java.util.HashMap` but shows a more time-deterministic behavior. Since possessing a deterministic memory performance is essential for real-time applications, this example also illustrates the suitability of the presented approach for the field of application it is intented for.

The memory performance of the method `setup` shown below which is part of the Javolution [5] library was specified in JML. Since the setup method is used by `FastMap`'s constructors to create and initialize a new map, its memory performance is of particular relevance for determining the memory consumption of instances of this class. The specification cases can be verified almost automatically by the KeY system and could probably be verified completely automatically using a more advanced quantifier instantiation heuristic[4]:

```
/*@ public normal_behavior
  @  requires capacity <= (1 << R0) && capacity>=0;
  @  working_space \space(new Entry[1][1<<R0]) +
  @      (2+capacity)*\space(new Entry());
  @ also public normal_behavior
  @  requires capacity > (1 << R0) && capacity<1<<30;
  @  working_space
  @      \space(new Entry[2*capacity][1<<R0])+
  @      (2+capacity)*\space(new Entry());
  @*/
private void setup(int capacity) {
    int tableLength = 1 << R0;
    /*@ loop_invariant 1 << R0 < capacity ?
```

[4] A more sophisticated quantifier instantiation heuristics for KeY that could add to a higher automation degree of the discussed example is currently under development but was not yet used for this example.

```
      @     tableLength>=1 << R0 &&
      @     tableLength<2*capacity :
      @     tableLength == 1 << R0;
      @ decreases 1 << R0 < capacity ?
      @        2*capacity-2-tableLength : 0;
      @ assignable tableLength;
      @ working_space_single_iteration 0;
      @*/
    while (tableLength < capacity) {
        tableLength <<= 1;
    }
    int size = tableLength >> R0;
    _entries = (Entry[][]) new Entry[size][];
    int i = 0;
    /*@ loop_invariant i>=0;
      @ decreases _entries.length-i;
      @ assignable i, _entries[*];
      @ working_space_single_iteration
      @         \space(new Entry[1 << R0]);
      @*/
    while (i < _entries.length) {
        int blockLength = 1 << R0;
        _entries[i++] =
            (Entry[]) new Entry[blockLength];
    }
    _head = new Entry();
    _tail = new Entry();
    _head._next = _tail;
    _tail._previous = _head;
    Entry previous = _tail;
    i = 0;
    /*@ loop_invariant i>=0 && previous!=null;
      @ decreases capacity-i;
      @ assignable _tail._next, i;
      @ working_space_single_iteration
      @         \space(new Entry());
      @*/
    while(i++ < capacity) {
        Entry newEntry = new Entry();
        newEntry._previous = previous;
        previous._next = newEntry;
        previous = newEntry;
    }
}
```

This demonstrates the potential of the presented approach even for rather complex code in realistic real-time applications.

Remark 61 *Given properties for a concrete JVM one could simplify the performance specification shown in the example. For instance with the JVM parameters we assume (see section 3.2) the working space expressions*

```
\space(new Entry[1][1<<R0])+
   (2+capacity)*\space(new Entry())
```

can be simplified to a better human readable

```
240+capacity*40
```

This simplified expression does not need to be computed "by hand" but can be retrieved from the KeY proof for the corresponding contract.

7 Conclusion

Formal software verification can effectively and in many cases also automatically ensure the correctness of memory performance constraints even for non-trivial applications as shown in section 6.2. Section 3.2 presented improvements and extensions to JML. One of these extensions makes it possible to specify the worst case memory consumption of loop iterations in a way that is verifiable by the KeY prover and also usable by it for determining the memory usage of the entire loop. Using KeY, memory performance contracts are verifiable in a modular way making their verification efficient and adaptable to software changes.

8 Future Work

Currently all specifications described in this work have to be provided by the user. It would be preferable to derive some of them, such as, for instance, loop invariants [16] or the memory consumption figures of single loop iterations, automatically by means of static analysis. Data flow analysis could help to provide likely memory usage estimation that does not necessarily need to be correct because they can then be formally verified as described in this work.

It would also be interesting to incorporate support for RTSJ [4] features, first of all scoped and immortal memory, in the JAVA CARD DL calculus since real-time and safety critical systems are, beside smart card applications, apparently the most likely field of application for formal verification of performance specifications.

Acknowledgements

I would like to thank Prof. Dr. Peter H. Schmitt and Mattias Ulbrich for their advices and suggestions contributing to this work.

References

1. R. Atkey. Specifying and verifying heap space allocation with JML and ESC/Java2 (preliminary report). In *Workshop on Formal Techniques for Java-like Programs (FTfJP)*, July 2006.
2. B. Beckert, R. Hähnle, and P. H. Schmitt, editors. *Verification of Object-Oriented Software: The KeY Approach*. LNCS 4334. Springer-Verlag, 2007.
3. W. S. Beebee, Jr. and M. Rinard. An implementation of scoped memory for real-time Java. *Lecture Notes in Computer Science*, 2211:289–??, 2001.
4. G. Bollella and J. Gosling. The real-time specification for Java. *Computer*, 33(6):47–54, 2000.
5. J.-M. Dautelle. The javolution homepage, `http://www.javolution.org/`.
6. C. Engel. A Translation from JML to JavaDL. Studienarbeit, University of Karlsruhe, Department of Computer Science, 2005.
7. P. Giambiagi and G. Schneider. Memory consumption analysis of java smart cards. In *Proceedings of XXXI Latin American Informatics Conference (CLEI 2005)*, page 12, Cali, Colombia, 2005.
8. D. Harel. Dynamic logic. In D. Gabbay and F. Guenther, editors, *Handbook of Philosophical Logic*, volume 2 of *Extensions of Classical Logic*, pages 497–604. D. Reidel Publishing Company, 1984.
9. I. J. Hayes and M. Utting. A sequential real-time refinement calculus. *Acta Informatica*, 37(6):385–448, 2001.
10. C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, 1969.
11. J. J. Hunt, F. B. Siebert, P. H. Schmitt, and I. Tonin. Provably correct loops bounds for realtime java programs. In *JTRES '06: Proceedings of the 4th international workshop on Java technologies for real-time and embedded systems*, pages 162–169, New York, NY, USA, 2006. ACM Press.
12. J. Krone, W. F. Ogden, and M. Sitaraman. Modular verification of performance constraints. In *ACM OOPSLA Workshop on Specification and Verification of Component-Based Systems (SAVCBS)*, pages 60–67, 2001.
13. G. T. Leavens, E. Poll, C. Clifton, Y. Cheon, C. Ruby, D. R. Cok, P. Müller, J. Kiniry, and P. Chalin. JML Reference Manual. Department of Computer Science, Iowa State University. Available from `http://www.jmlspecs.org`, Feb. 2007.
14. B. Meyer. Applying "Design by Contract". *Computer*, 25(10):40–51, 1992.
15. J. Rumbaugh, I. Jacobson, and G. Booch, editors. *The Unified Modeling Language reference manual*. Addison-Wesley Longman Ltd., Essex, UK, UK, 1999.
16. P. H. Schmitt and B. Weiß. Inferring invariants by symbolic execution. Submitted to 4th International Verification Workshop (VERIFY'07), Workshop at 21st Conference on Automated Deduction (CADE-21), Bremen, Germany, 2007.