Specifying and Verifying Heap Space Allocation with JML and ESC/Java2 (Preliminary Report)

Robert Atkey

LFCS, School of Informatics
University of Edinburgh
bob.atkey@ed.ac.uk

Abstract. We examine JML's support for specifying the heap space allocation of Java programs. In this report we restrict ourselves to specifying and verifying only allocation but not de-allocation. We identify some problems with with JML's support and suggest alternatives. Also, we describe an implementation of heap space allocation verification in ESC/Java2. This implementation has been tested on small examples.

1 Introduction

This report is concerned with the specification and verification of the heap space consumed by Java programs. Control of heap space usage is one part of the general problem of ensuring that programs do not consume more resources than they are allocated while also ensuring they successfully execute. Resource constrained computing — by resources we mean things like memory space, processor time and usage of external interfaces such as disks and networks — is important both to very small devices such as smartcards or mobile phones where the total amount of resources in limited, and to very large multiple-user systems such as database servers and grid computing clusters on which no one user should be allowed to monopolise the machine.

By "heap space" we mean the memory allocated by the creation of new Java objects and arrays. Java does not have any facility for explicit deallocation of memory and relies on a garbage collector to free up unused memory. The specifications of heap space in this paper will not assume that heap space is ever freed since we can not guarantee when or if the garbage collector will run.

We have investigated the specification of heap space allocation for Java programs in JML, and its verification in ESC/Java2. JML, the Java Modelling Language [12, 13], is a language for specifying the behaviour of Java classes. Specifications are included in Java source code as annotations embedded inside specially formatted comments. JML is able to express pre- and post-conditions on methods and class invariants, in an Eiffel-like Design By Contract style [15]. Logical statements within JML specifications are written in a Java-like syntax extended with logical features. Java methods declared as `pure` which have no side-effects may be used in annotations.

JML includes features for specifying the heap space consumption of methods as well as for accessing the heap space specification of other methods and the

run-time sizes of objects. The current facilities are based on a paper by Krone, Ogden and Sitaraman [11]. We present JML's current design in Section 2 with an example illustrating its use.

We have identified several criticisms of JML's facilities for heap space allocation specification and verification which we set out in detail in Section 3. In summary, these are:

- Poor interaction between the reflection of methods' heap space allocation specifications into specification expressions and program state (Section 3.1).
- The inability to access the current heap space allocation from specification expressions within a method. Such a facility is vital for verifying methods (Section 3.2).
- Problems with accessing the heap sizes of run-time data, particularly that of objects with non-pure constructors (Section 3.3).

We propose alternatives to JML's current design to address these problems.

ESC/Java2 [7] is an automatic tool for checking Java source code annotated with JML. It is a development of the original ESC/Java developed by Compaq Research [10]. It checks for the absence of some runtime errors such as array out-of-bounds indexing or null dereferencing, and checks some user assertions written in a subset of JML. ESC/Java2 extends ESC/Java by accepting all of JML and expanding the range of features checked.

We have extended ESC/Java2 to support the checking of annotations relating to heap space allocation and the reflection of objects' run-time sizes into JML specification expressions. We have also partially implemented support for reflecting methods' heap space into specification expressions. Our implementation is enough to expose the problems we describe in Section 3.1 and our proposed solution makes use of existing JML features already present in ESC/Java2. The implementation is described in Section 4.

Overview In Section 2 we describe JML's current design for specifying heap space allocation, with an example. We also discuss desirable features of such a design. We highlight some deficiencies of JML's design in Section 3 and suggest alternatives. We then describe our changes to ESC/Java2 in Section 4. Finally, in Section 5 we summarise our work and suggest further directions.

2 JML's support for Heap Space specification

JML's current design [13] includes a specification clause for method declarations that sets an upper bound on the amount allocated by the method's execution:

```
//@ ...
//@ working_space <long-expr> [if <bool-expr>];
//@ ...
public int methodName (...) { ...
```

As the names suggest, `<long-expr>` and `<bool-expr>` stand for expressions of type long and boolean respectively. The `if <bool-expr>` part is optional. All the expressions in these clauses are evaluated in the post state of the method, but selected sub-expressions can be evaluated in the pre-state using the `\old(<expr>)` construct. The semantics of `working_space` specifies that, if `<bool-expr>` holds, terminating execution of the method (including any methods it calls) should allocate no more than the value of `<long-expr>` bytes of heap space. This includes exceptional termination, but there is no way to determine the exception thrown, except by making a separate heavyweight specification case for each exception.

JML also includes two specification expression functions for heap space: `\working_space` and `\space`. The function `\working_space(...)` takes as an argument a quoted method call expression and evaluates to a value of type long. For example: `\working_space(obj.m (a, b))`. The method call "`obj.m(a, b)`" is not evaluated, but quoted. The whole expression evaluates to the value of `m`'s `working_space` clause given the static type of obj, the values of `a`, `b` and `obj` and the current program state. JML does not specify what the value should be if there is more than one `working_space` clause, but we take it as the minimum of the applicable clauses. The function `\space(...)` takes an expression of reference type (i.e. an object or an array) and evaluates to the amount of heap space occupied by the referenced object. This does not include the heap space occupied by objects linked from the starting object. The result is of type `long`.

The design of these facilities has been adapted from the work of Krone *et al* [11]. In that paper the authors argue for several points to be taken into consideration when designing a specification language for resource usage, including: the usage of actual program values when specifying resource usage, the importance of modularity of specification and the ability to do verification.

The use of actual program values in specification expressions, in the place of metrized versions of values such as the length of a list data structure or the depth of a tree data structure, helps ensure precision in the specification of resource usage. For example, the heap space usage of a method operating on a list may depend on the contents of the list, not just its length. JML supports this by allowing references to any variable in scope in the post-state in `working_space` clauses.

Modular specification of components is a necessity for large programs. A change in the specification for one component should be automatically carried through to all parts that refer to it. In the case of resource usage specifications, this means that methods' resource usage specifications should not hard-code the resource specifications of methods they use. Rather, they should refer to the resource usage by the method's name and the arguments it will be passed. When the called method is altered, the resource specification of dependent methods will be automatically updated. JML attempts to support this by means of the `\working_space` functions. In Section 3.1, we show that this mechanism does not work in the presence of state changing functions and suggest a simpler method.

Krone *et al* also emphasise the need for verification. They give verification rules for loops and procedure calls that incorporate resource usage. JML does not currently have support for verifying the heap space usage of loops. We address this in Section 3.2.

2.1 Example

The following Java source code illustrates the use of JML's facilities for specifying heap space allocation:

```
class ResourcesExample {
    //@ working_space \space(new int[2]);
    /*@ pure @*/ ResourcesExample () { new int[2]; }

    //@ working_space \space(new ResourcesExample())
    //@              + \working_space(new ResourcesExample ());
    public static void fiddle () { new ResourcesExample (); }

    //@ requires c >= 0;
    //@ working_space c * \working_space (fiddle ());
    public static void muddle (int c) {
        for (int i = 0; i < c; i++) fiddle ();
    }
}
```

This class defines a single constructor and two static methods:

`ResourcesExample` The constructor does two things: it makes an implicit call to `Object`'s constructor and it allocates a two element array of `int`s. Assuming that the constructor of `Object` is declared to not allocate any memory, the total `working_space` of this constructor is the space occupied by a two element array of `int`. The space occupied by the newly allocated object is not included. We have had to declare the constructor as `pure` in order to use it in the specification of `fiddle`. Obviously, not all constructors can be declared `pure`. We address this problem in Section 3.3.

`fiddle` This static method creates a new instance of `ResourcesExample`. Therefore, the heap space allocated by this method is the space required by an instance of `ResourcesExample` and the space required by the execution of the constructor. The expression `new ResourcesExample ()` in the specification is only used as a guaranteed way to obtain a non-null `ResourcesExample` reference. The constructor was required to be declared `pure` so it could be legally used within a specification expression.

`muddle` The static method takes an `int` argument `c` and calls the `fiddle ()` method `c` times, hence its `working_space` is `c` times the working space of `fiddle ()`. This specification demonstrates the use of `\working_space` to obtain the heap space allocation specification of another method.

3 Criticisms of JML's support

We have identified several shortcomings of JML's facilities for heap space specification and verification. In this section, we describe them and offer some possible solutions.

3.1 Interaction between the program state and \working_space

The heap space consumption of a Java method may depend on the state of variables external to the method. Hence, the function \working_space may have different values for identical quoted invocations depending on the program state that it is evaluated in. The working_space specification clause is defined by JML to be evaluated in the post-state of the method, and it can access the pre-state by use of the \old(...) function. The restriction to only the pre- and post-state means that it is impossible to use the \working_space function as intended, as the following example demonstrates:

```
class StatefulResourcesExample {
    boolean flag;

    //@ working_space flag?\space (new int[2]):0;
    public void condAlloc () { if (flag) new int[2]; }

    //@ working_space 2*\working_space(condAlloc()) if \old(flag);
    //@ working_space \space (new int[2]) if !\old(flag);
    public void tricksy () {
        condAlloc ();
        flag = !flag; condAlloc ();
        flag = !flag; condAlloc ();
    }
}
```

This class has one boolean instance variable flag, and two methods:

condAlloc This method allocates a two element array of int if flag is true. Otherwise, it does no allocation.

tricksy This method calls condAlloc three times, inverting flag between each call. In terms of heap space allocation there are two possibilities: if flag is true to start with then the method will allocate two two-element arrays of int, if flag is false to start with then the method will allocate one two-element array. These two cases are covered by two working_space clauses. In the first case we have access to a state in which flag is true (either the pre- or post- state) and so we can specify the behaviour in terms of the working space of condAlloc. In the second case we do not have access to a state where flag is true and so we must resort to specifying the space directly.

There is no way to specify the heap space consumption of `tricksy` without referring directly to `condAlloc`'s actual heap space allocation. Since there is no way in JML of evaluating a specification expression in an arbitrary program state, it seems likely that `\working_space` could never be made to interact well with methods whose heap space consumption depends on the program state.

In order to retain modularity of heap space consumption specifications, we propose that auxiliary model functions be used to specify the heap space allocation of methods. For the example above we need a function with the following specification and signature:

```
//@ ensures \result == (f?\space(new int[2]):0);
//@ signals (Exception) false;
/*@ pure @*/ public int condAlloc_space (boolean f)
```

This function explicitly takes the relevant part of the state and returns the working space specification required. Unfortunately, this specification is not implementable in Java since we have no access to the value of `\space(new int[2])`. This is not a severe problem since we can give a dummy implementation and skip its verification. Alternatively, it would be possible to provide a class containing static methods corresponding to the `space` function at different types which is specified to have the same values as the specification functions. Each JVM implementation would have to provide an implementation of this class to supply the real values for run-time specification checking.

This problem notwithstanding, the specifications of the methods can be replaced with:

```
//@ working_space condAlloc_space (flag);
public void condAlloc () { ... }

//@ working_space 2*condAlloc_space(flag)+condAlloc_space(!flag);
public void tricksy () { ... }
```

Thus the working space of `tricksy` has been specified directly in terms of the working space of `condAlloc`. If the heap space allocation of `condAlloc` were to change then the definition of `condAlloc_space` would change and so would the specification of `tricksy`. Thus, this alternative technique is modular, unlike the implicitly state-dependent `\working_space` functions. This technique also has the advantage of reusing existing JML features.

3.2 Accessing the current heap space allocation

In [11] Krone *et al* augment the usual loop invariant and variant annotations with information on the maximum amount of heap space allocated by the iterations of the loop. This information is vital for verifying the heap space consumption of methods that use loops. JML currently has no way of specifying this information.

We fix this by adding a special variable for specification expressions, named `\current_working_space`. This special variable denotes the current amount of

heap space allocated by the current method above that allocated by its callers. It may only occur in specification expressions within methods. We do not need to make a distinction between the maximum heap usage and the current heap allocation as Krone *et al* do because Java has no de-allocation operation to explicitly decrease the size of the heap.

An example of its use is taken from the body of the `muddle` method above:

```
//@ loop_invariant 0 <= i && i <= c;
//@ loop_invariant
//@    \current_working_space <= i * \working_space (fiddle ());
for (int i = 0; i < c; i++)
    fiddle ();
```

This annotation states that the heap space allocated by the loop is always equal to `i` times the heap space allocation of the `fiddle()` method. Our extension to ESC/Java2, described in Section 4, can use this annotation to verify that this method does indeed match its specification[1].

The advantage of exposing the current heap allocation via a special variable rather than an additional loop annotation is that it may also be used in other method body annotations such as `assert`, allowing the programmer to document the expected heap consumption behaviour of their program and to help find mismatches between implementation and specification.

3.3 Accessing the sizes of run-time data

In the example in Section 2.1, it was necessary to declare the constructor `pure` so that it could be legally used in a JML specification expression. For the purposes of the specification we were not interested in the effect of the constructor, nor even its specification, merely that it provide us with a non-null reference of the correct type so that the `\space` function would return the amount of space it occupied. The requirement of purity is a serious limitation since we would like to be able to access the sizes of objects that have non-pure constructors, yet JML does not currently have a construct to obtain run-time object sizes without evaluating some specification expression of the correct type.

We propose a fix by adding to JML a way of obtaining the run-time space consumption of data on the heap by just their type in the case of single objects and by their type and size in the case of arrays. This is accomplished by removing the `\space` function and replacing it by two functions `\object_space` and `\array_space`.

The function `\object_space` takes a single argument of type `\TYPE`, the JML type of Java types, and returns the size in bytes occupied by objects of the supplied type at run-time as a value of type `long`, which is always greater than

[1] Proviso: this verification is only: (a) safe when the `-loopSafe` switch is used; and (b) possible when the axiom $\forall x.\forall y.xy + y = (x + 1)y$ is added to help the theorem prover Simplify prove the verification condition.

or equal to 0. If the supplied type is primitive, an array type, an interface type or an abstract class type then the function evaluates to 0.

The function `\array_space` takes two arguments: one of type `\TYPE` and one of type `long`. It returns a value of type `long`: the size in bytes occupied by an array of the specified type with the specified number of elements, which is always greater than or equal to 0. For any two reference types `ty1` and `ty2` (i.e. when `ty1` and `ty2` are array, interface or object types), the values of `\array_space(ty1, n)` and `\array_space(ty2, n)` are equal for all `n`.

Sometimes the `\space` function is useful for specifying the amount of heap space allocated, if the type of the object is not known at compile-time. An example is an abstract clone method; when specifying such a method we do not know the type of the implementing class, so we do not know what type to pass to `\object_space`. In Section 4.3 we show how to axiomatise `\space` in terms of our primitive space functions.

4 Extending ESC/Java2

We have extended ESC/Java2's JML support to cover heap space allocation annotations, both with support for the JML functions `\space` and `\working_space` (though this is limited, see below) and with support for our alternative functions. The current release already parses and type-checks `working_space` clauses, so our work focused on implementing them in the later phases of ESC/Java2's operation.

Our modifications involve three parts: extra functions and axioms for the logic of ESC/Java2 to symbolically interpret the run-time sizes of object instances and arrays; an extension of the translation into guarded commands to support the verification of heap space consumption specifications; and extending the translation of specification expressions into the logic to handle the `\space`, `\array_space` and `\object_space` functions, the `\current_working_space` variable and an encoding of methods' working space clauses into logical axioms to support the `\working_space` function.

4.1 Extending the logic

Two new functions *object_space*, of arity 1, and *array_space*, of arity 2, have been added, with axioms:

$$\forall t.\mathit{object\_space}(t) \geq 0$$
$$\forall t.\mathit{object\_space}(\mathit{array}(t)) = 0$$
$$\forall t.\forall n.\mathit{array\_space}(t, n) \geq 0$$
$$\forall t.\forall n.t <: \texttt{java.lang.Object} \Rightarrow \mathit{array\_space}(t, n) = \mathit{array\_space}(\texttt{java.lang.Object}, n)$$

To this for each primitive type `primtype` we also add an axiom of the form $\mathit{object\_space}(\texttt{primtype}) = 0$. The function *array* constructs an array type from

any other type, so for all arrays *object_space* is defined to be 0. The relation <: represents Java subtyping and `java.lang.Object` represents that Java type in the logic. Java regards all array types as well as all object types as subtypes of `java.lang.Object`. ESC/Java2 currently has no way of distinguishing abstract class and interface types from instantiable object types, so we could not implement this part of our specification of *object_space*, but it would be an easy extension.

The axioms evidently express the requirements on the two functions described in Section 3.3. By axiomatising in this way, rather than supplying concrete values, we avoid tying ourselves to any particular JVM implementation.

4.2 Accounting for and Verifying Heap Space Consumption

After parsing and typechecking JML-annotated Java source code, ESC/Java2 translates each method into a guarded command language, as described in [14]. The method's preconditions are inserted as assumptions at the start of the guarded command sequence and the post-conditions are asserted at the end. It also translates the annotations on the rest of the program into a "background predicate" detailing the other classes, methods and fields in the program and their specified behaviour. The guarded command language is given a semantics by a weakest liberal precondition function. ESC/Java2 generates an approximation of the weakest liberal precondition for the method and passes it to a theorem prover to check that it is consistent with the background predicate. By default, this prover is the automatic first-order theorem prover Simplify [9].

In order to verify `working_space` annotations on methods we have altered the translation to guarded commands. We maintain a count of the current number of bytes allocated by the execution of the method in a variable called `WSPC`. At the start of the guarded command sequence it is initialised to 0. Each Java source level expression that causes allocation of memory is translated to a sequence of guarded commands that includes an instruction for incrementing `WSPC`. At the end of the guarded command sequence, the value of `WSPC` is checked against the stated `working_space` clauses for the current method, in the same manner as the checking of post-conditions.

There are two ways of constructing new arrays in Java: by an array literal or the allocation of an array with default values of given type and dimensions. For both of these, the `WSPC` variable is incremented by $\mathit{array\_space}(t, n)$ for the appropriate type t and length n. For multi-dimensional arrays, the space for the sub-arrays is summed and added to the space for the containing array. For object instantiation, the `WSPC` counter is incremented by $\mathit{object\_space}(t)$ for the appropriate type t. This is done after the constructor call to ensure correct behaviour in the event that the constructor raises an exception.

Incrementing `WSPC` for method calls is more involved. First, we must extract the `working_space` clauses from the specification of the method called and the methods it overrides in super-classes. This is accomplished with a minor extension of the generation of specifications for method calls as described in Section 7 of [14]. For each method, this generates a list of (P, W) pairs. The predicate

P is a pre-condition derived from the `if` part of the `working_space` clause and the method's preconditions. The expression W is the translation of the value of `working_space`. During the translation of a method call, after the program state has been updated, a series of guarded command assume statements is inserted of the form $P \Rightarrow \texttt{WSPC'} \leq W$, where `WSPC'` is a fresh variable. An assumption that $\texttt{WSPC'} \geq 0$ is also inserted. The variable `WSPC` is then incremented by `WSPC'`. This translation ensures that the current heap size is incremented by a value bounded by the `working_space` specification of the called method. If the called method has no specification then the possible allocated memory is unbounded.

4.3 Interpreting Specification Expressions

Our JML specification expression functions `\object_space` and `\array_space` are directly translated into their counterparts in the logic. Interpreting our new variable `\current_working_space` is also straightforward; it translates directly to the variable `WSPC`. This variable, as described above, holds the current count of bytes allocated by the method's execution to this point, as required.

We translate `\space` to a function *space* in the logic. We axiomatise the behaviour of *space* in terms of *object_space* and *array_space*:

$$\forall x.x = \texttt{null} \Rightarrow space(x) = 0$$

$$\forall x.x \neq \texttt{null} \wedge typeof(x) = array(elemtype(typeof(x))) \Rightarrow space(x) = array\_space(elemtype(typeof(x)), arrayLength(x))$$

$$\forall x.x \neq \texttt{null} \wedge typeof(x) \neq array(elemtype(typeof(x))) \Rightarrow space(x) = object\_space(typeof(x))$$

These axioms state that: if the reference is null, then the value is 0; if the reference points to an array (the dynamic type given by *typeof* is of an array type) then the value is equal to the appropriate value of *array_space*; otherwise it is equal to the appropriate value of *object_space*. The formulation $typeof(x) = array(elemtype(typeof(x)))$ is the standard way of distinguishing values of array type in the logic of ESC/Java2. Note that the type-checking phase of ESC/Java2 ensures that we only apply *space* to values of reference type.

Our interpretation of the `\working_space` function is currently limited to invocations of methods that are declared `pure` and whose `working_space` clauses are non-overlapping. We have augmented ESC/Java2's axiomatisation of `pure` methods to also include information about their heap space allocation.

The axiomatisation is described in detail by Cok in [6]; we give a short summary here. Given an invocation of a method `C.m` in a specification expression, ESC/Java2 translates this to a function application for some function name derived from the class and method name with the current state, the instance object (if applicable) and the other arguments as parameters. It then also inserts assumptions of the form $\forall \overrightarrow{x}.P \Rightarrow Q$, where $\overrightarrow{x}$ are the formal arguments (including the object instance in the case of non-static methods), P is the method's pre-condition and Q is the method's post-condition with `\result` translated to the

chosen function name[2]. This axiomatisation is done once for each state that the method is called in. Note that the state argument is not quantified over: we have a new axiom for each possible state.

To handle `\working_space` we add assumptions of the form

$$\forall \overrightarrow{x}.P \Rightarrow C.m\#WSPC(state, \overrightarrow{x}) = W$$

for each (P, W) derived from the method's `working_space` clauses. Each instance of `\working_space` quoting a call of `C.m` is translated to an application of $C.m\#WSPC$ with the required parameters.

As pointed out by Darvas and Müller [8], ESC/Java2's axiomatisation of `pure` methods leads to unsoundness in the case of unsatisfiable specifications and does not handle weak purity. As a consequence of adapting ESC/Java2's current axiomatisation, our implementation of `\working_space` suffers from the same problems, as well as not being able to deal with methods that update global state. It is possible to adapt Darvas and Müller's solution to handle some cases of `\working_space`, though this would not directly work for recursive methods or methods that throw exceptions. However, given the problems inherent in the design of this function we feel that it is not necessary.

4.4 Verification of Heap Space Allocation with ESC/Java2

We have verified the examples in this paper using our modified ESC/Java2. Verifying methods without loops is straightforward. Verifying methods with loops requires annotation of the loops with invariants using `\current_working_space`. Simplify is not always able to prove the resulting verification conditions, but it is possible to add extra axioms to the background predicate to help Simplify. ESC/Java2 is currently undergoing extension to be able to use more powerful interactive provers. This will allow the verification of more elaborate programs.

In our experience, there are two main difficulties with verifying the heap space usage of larger programs. Firstly, any Java program of reasonable size makes heavy use of library classes and so we require the specification of the heap space allocation of library functions. The documentation almost never gives the required level of detail to specify this information.

Secondly, most Java programs rely heavily on the garbage collector to ensure good heap space usage. Since Java has no explicit de-allocation instruction it is difficult to verify methods which rely on the de-allocation of heap space. We see two possible approaches. One is to make the programmer responsible for manually re-using memory. This is the method used to simulate de-allocation in the Camelot language of the MRG project [1]. We have attempted to implement such a technique with our extended ESC/Java2; however, we ran into problems with difficulties with the reasoning about linked data structures required (such as the free list), and the fact that the arithmetical verification conditions generated

[2] The translation differs from the one described in [6] since it does not handle `signals` and `diverges` clauses. These have not been implemented in ESC/Java2's axiomatisation of methods.

are beyond Simplify's capabilities. Alternatively, JML could be augmented with a system capable of identifying unique references; when such references are lost, then the associated memory could be regarded as de-allocated. This relies on the garbage collector being able to collect all unreferenced objects.

5 Conclusions

We have presented a description of JML's current support for specifying heap space allocation and several criticisms of that support. In summary our criticisms are that: the reflection of methods' heap space specifications does not interact well with methods that alter the program state; it is not possible to access the current heap space allocated from within a method, which is vital for verification; and it is often not possible to obtain the run-time sizes of objects. We have offered solutions to these problems. We have also presented a preliminary implementation of heap space allocation verification in ESC/Java2.

Barthe, Pavolva and Schneider [2] have described an extension to the Bytecode Modelling Language for specifying and verifying heap space allocation bounds for Java bytecode programs. As with our approach, they use an auxillary variable to track allocation. They do not consider the abstract specification of allocation bounds; rather, they specify the bounds directly in terms of their auxillary variable. They also give an algorithm for inferring heap space allocation bound specifications for methods.

We put forward the following as directions for future research:

Non-terminating methods JML currently defines that the `working_space` clauses must hold on termination of a method. This is a consequence of evaluating the clause in the post-state of the method. Non-terminating methods, such as server threads that loop processing requests, are also candidates for heap space usage specification – one does not want a server to suffer from a slow memory leak. A possible solution is to require that the `working_space` clause hold for all executions of the method, non-terminating or otherwise. This would require evaluation in the pre-state.

De-allocation As mentioned above, Java does not include any facilities for explicit de-allocation. An alternative is to use a linear type system such as that of [3] to determine when an object is capable of being garbage collected and treat that as de-allocation. This is the approach taken by [5]. Specification of methods that de-allocate memory will require two values: the amount of memory they require above their caller, and the amount of memory of they free. Verification will require two variables: one to track the difference in heap size from the start of the method, and one containing the maximum heap size required. In order to maintain a direct connection to actual run-time performance, we must have some guarantee that the memory is actually de-allocated when we expect it to be. This will require changes to the JVM.

Other resources We have not discussed the specification and verification of other resources such as network usage and CPU time in this paper. Chander *et al* [4] discuss enforcement of resource bounds using an acquire/consume

paradigm and verify them using ESC/Java. The design of JML includes facilities for specifying the time consumed by the execution of a method (in terms of the number of JVM instructions executed), in a manner similar to the support for specifying heap space usage. However, the `\duration` functions, which offer the same service as the `\working_space` functions, suffer the same problems. Also, it is not possible to obtain the number of JVM instructions for an arbitrary piece of Java code.

Acknowledgement This work was funded by the ReQueST grant (EP/C537068) from the Engineering and Physical Sciences Research Council.

References

1. David Aspinall, Stephen Gilmore, Martin Hofmann, Donald Sannella, and Ian Stark. Mobile resource guarantees for smart devices. In Gilles Barthe, Lilian Burdy, Marieke Huisman, Jean-Louis Lanet, and Traian Muntean, editors, *CASSIS 2004*, volume 3362 of *LNCS*, pages 1–26, January 2005.
2. G. Barthe, M. Pavlova, and G. Schneider. Static analysis of memory consumption using program logics. In *3rd IEEE International Conference on Software Engineering and Formal Methods*. IEEE, IEEE Computer Society Press, September 2005.
3. John Boyland, James Noble, and William Retert. Capabilities for aliasing: A generalisation of uniqueness and read-only. In Jørgen Lindskov Knudsen, editor, *ECOOP 2001 — Object-Oriented Programming, 15th European Conference*, volume 2072 of *LNCS*, pages 2–27, 2001.
4. Ajay Chander, David Espinosa, Nayeem Islam, Peter Lee, and George Necula. Enforcing resource bounds via static verification of dynamic checks. In Mooly Sagiv, editor, *14th European Symposium on Programming, ESOP 2005*, volume 3444 of *LNCS*, pages 311–325, 2005.
5. Wei-Ngan Chin, Huu Hai Nguyen, Shengchao Qin, and Martin Rinard. Memory usage verification for OO programs. In *Static Analysis: 12th International Symposium, SAS 2005*, volume 3672 of *LNCS*, pages 70–86, September 2005.
6. David R. Cok. Reasoning with specifications containing method calls in JML and first-order provers. In *6th Workshop on Formal Techniques for Java-like Programs - FTfJP2004*, June 2004.
7. David R. Cok and Joseph R. Kiniry. ESC/Java2: Uniting ESC/Java and JML. In Gilles Barthe, Lilian Burdy, Marieke Huisman, Jean-Louis Lanet, and Traian Muntean, editors, *CASSIS 2004*, volume 3362 of *LNCS*, pages 108–128, January 2005.
8. Ádám Darvas and Peter Müller. Reasoning about Method Calls in JML Specifications. In *7th Workshop on Formal Techniques for Java-like Programs - FTfJP2005*, July 2005.
9. David Detlefs, Greg Nelson, and James B. Saxe. Simplify: A theorem prover for program checking. Technical Report HPL-2003-148, HP Laboratories, July 2003.
10. C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended static checking for Java. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation (PLDI'02)*, pages 234–245. ACM Press, June 2002.

11. Joan Krone, William F. Ogden, and Murali Sitaraman. Modular verification of performance constraints. Technical Report RSRG-03-04, Department of Computer Science, Clemson University, May 2003.
12. Gary T. Leavens, Albert L. Baker, and Clyde Ruby. Preliminary Design of JML: A Behavioral Interface Specification Language for Java. Technical Report TR #98-06-rev29, Department of Computer Science, Iowa State University, January 2006.
13. Gary T. Leavens, Erik Poll, Curtis Clifton, Yoonsik Cheon, Clyde Ruby, David Cok, Peter Müller, and Joseph Kiniry. JML reference manual. `http://www.jmlspecs.org/`, January 2006.
14. K. R. M. Leino, James B. Saxe, and Raymie Stata. ESCJ 16c: Java to guarded commands translation. Available from the ESC/Java2 website, August 1998.
15. Bertrand Meyer. *Object-oriented Software Construction*. Prentice Hall, 2nd edition, 1997.