Specification and Verification of Garbage Collector by Java Modeling Language

Wenhui Sun, Yuting Sun, Zhifei Zhang
Department of Computer Science and Technology
Beijing Jiaotong University
Beijing, China
{whsun1, ysun, zhfzhang}@bjtu.edu.cn

Jingpeng Tang
Department of Computer Science
Utah Valley University
Oren, Utah
JTang@uvu.edu

Kendall E. Nygard, Damian Lampl
Department of Computer Science
North Dakota State University
Fargo, ND, USA
{kendall.nygard,damian.lampl}@ndsu.edu

***Abstract*— The Java garbage collector effectively avoids some security holes and improves the utilization rate of resources. Guaranteed reliability of the garbage collector is a challenge due to the complexity of the interaction between the collector and the user program; the highly abstracted garbage collector algorithms cannot reflect the real implementation details. System complexities have allowed dynamic analysis based on Design by Contract (DBC) to become an important method for ensuring software quality. Java Modeling Language (JML) inherits all the advantages of contractual design, and became a behavior interface specification language for Java. JML can be used to regulate module behavior and detailed design of Java programs. In this paper, we discuss the JML specifications for the functional requirements of the garbage collector in Hoare-style. This approach can improve the reliability and correctness of the software system in the extent of real environments and run-time.**

Keywords- design by contract, Java Modeling Language, garbage collector.

I. INTRODUCTION

A large number of existing research works have addressed validating various types of garbage collection algorithms. However, these works focus on highly abstract algorithms with little work on implementations. Birkedal, Torp-Smith, and Reynolds gave an informal proof of a copying garbage collector [1]. Russinoff mechanically verified an incremental garbage collector under abstracted memory and a user program without actual environmental and implementation details. Validated algorithms are not equivalent to executable programs [2]. Lin, Chen, and Li verified the incremental stop-the-world mark-sweep garbage collector, which is more complex [3]. However, the interactions between a user program and garbage collector do not typically exist for a stop-the-world garbage collector.

Java Garbage Collector is an important component of a software system that can effectively avoid dangling pointer bugs, memory leaks, double free bugs, and can improve memory utilization [4]. Multi-threading makes possible for a concurrent incremental garbage collector. However, compared to a stop-the-world garbage collector, it is more complex and the reliability issue is more challenging. Java is an object-oriented language with inheritance, polymorphism, and dynamic binding properties. The program execution is no longer simply based on static typing and must now accommodate dynamic typing, meaning the type will not be known until execution time. This will introduce extra complexity to ensure program correctness. If the garbage collector process produces errors or exceptions, the user program will run into unpredictable consequences. Therefore, ensuring the reliability of the garbage collector is extremely important.

Many practitioners utilize Design by Contract (DBC) to improve software quality. Java Modeling Language (JML) is a DBC implementation in Java, and is also a precise formal specification language for Java programs [5][6]. JML can accurately describe functional requirements and generate efficient testing cases which can avoid ambiguity and inaccuracies caused by natural language [7]. Formal interface specifications written in JML can also encourage automated testing.

This paper discusses the validation and verification of an incremental mark-sweep garbage collector. The security interaction between the garbage collector and the user program is accurately described by applying JML precondition, postcondition, and invariants in Hoare-style logic. The JML runtime assertion will automatically perform formal verification to ensure the correctness of the garbage collector. This study focuses on real environment memory objects. The JML specification covers both normal and abnormal behavior, which can accurately describe the real-time environment. The assertions are runtime execution, thus they can effectively handle polymorphic, inheritance and dynamic binding for object-oriented software. In our approach, program execution is not only the result of a function generation process, but also an assertion checking process. This approach can improve correctness and

Copyright (c) IARIA, 2015. ISBN: 978-1-61208-389-6

reliability for the garbage collector, quickly position errors, and handle abnormal behavior during collection.

The main contributions of this paper are listed as follows:

(1) Using JML to verify the incremental mark-sweep garbage collector. JML, a Hoare-style syntax for pre- and postconditions and invariants, is a DBC implementation in Java. If the inputs meet the requirements, we should get the expected outputs. In more detail, if we take the parameters as inputs and returns as the outputs, then the responsibility of the caller (client) is to ensure that the correct parameters are provided, while the obligation of the supplier is to ensure the correct results are returned.

(2) Verifying the write barrier of the garbage collector using JML. The verification can avoid incorrect operation due to memory access and modification by the user program, to improve the correctness of the interaction between the garbage collector and the user program.

(3) Improve simplicity and understandability of the program function code. By separating the original program function code from the DBC checking code using JML, the program function code no longer mingles with DBC code block, thus avoiding unnecessary confusion. Also, the postcondition failure can easily locate errors. Improvement of algorithm reliability, as well as code understandability, can be achieved.

The rest of the paper is organized as follows: Section 2 describes JML and examples. Section 3 describes the garbage collector and write barrier algorithm using JML. Section 4 establishes the capabilities of the garbage collector in detail. Finally, Section 5 points out the conclusion and future work.

II. INTRODUCTION TO JML

A. *Features of JML*

JML specifications are written as Java annotation comments in the source files, and can be compiled with any Java compiler. These specifications are more abstract without logic implementation and thus can increase the modularity and accuracy of the source code [7][8]. By using DBC ideas, JML inherits all of its advantages, and is an excellent specification language:

(1) Documentation. JML provides semantics to formally specify interface, behavior, and detailed design. Java modules with JML specifications can be compiled with any Java compiler, making JML well suited for documenting reusable components, libraries, and frameworks [8].

(2) Clear Obligation. Pre- and postconditions separate the obligation. A precondition error indicates that the user's input does not meet the conditions, while a postcondition error indicates the procedural methods do not meet the requirement [9][10].

(3) High Efficiency. Since each execution of pre- and postconditions checks will consume resources, JML can turn off these checks to avoid unnecessary consumption of resources. This mechanism can decrease the cost of debugging and testing.

(4) Modular Reasoning. JML is abstract so that by reading the formal specification of a method its function is understood with no need to go inside other referenced methods (JML modular reasoning). JML modularity brings the benefits of easy understanding, but shields the details. The user will cannot understand the contents due to the lack of corresponding information.

In addition, the quantifier, specification inheritance, and pre-process can make the specification more accurate. Java modules with JML specifications can describe a method or class's anticipated behavior, without affecting the normal code while compiling. This can provide an early detection of incorrectness to improve the security of a system. Finally, Java modules with JML specifications can be compiled unchanged with any Java compiler. Various verification tools, such as a runtime assertion checker and the Extended Static Checker (ESC/Java) are available to aid the development. If the program does not implement the specification, JML throws an unchecked exception to explain that the program violates the specification.

B. *JML Syntax and Examples*

JML is a behavioral interface specification language for Java modules. JML provides semantics to describe the behavior of a Java module, preventing ambiguity with the module designers' intentions. Developers use JML to write classes and interfaces in the form of specifications. Each of the methods and interfaces written in accordance with the functional requirements is a JML formal specification. Developers must consider specific context of the system within which the method is running. The more precise the specification is, the more correctness will be achieved.

The example below illustrates JML usage and how it ensures reliability of a program. Assume class CustomerManager can manage all customer information of class BasicCustomerDetails. CustomerManager provides users with the add() method to create new customers. When a clear requirement of the add() method is available, we can develop JML specification as follows:

```
/*@ public normal_behavior
  @ public invariant count>=0;
  @    requires !theManager.idActive(theCustomer);
  @    assignable theManager;
  @    ensures
  @       theManager.count==\old(theManager.count+1);
  @          theManager.idActive(theCustomer);
  @*/
public void add(BasicCustomerDetails theCustomer);
```

Figure 1. JML example.

JML invariant assertion count is always greater than or equal to zero. Class CustomerManager's invariant count is

true under all circumstances. In line 3, the keyword requires starts the precondition followed by a precondition assertion,
@ requires !theManager.idActive(theCustomer);

The precondition has to be true; otherwise, the caller is not able to call this method. This shows that in order to legally call add() to add theCustomer, theCustomer.id should be inactive. This will be asserted during runtime. Keyword assignable can modify the variable theManager. Keyword ensures introduces the postcondition which should be true after the execution, otherwise there are errors in the implementation of the add() method. In this case, the postcondition includes two assertions,
@theManager.count==\old(theManager.count+1);
@theManager.idActive(theCustomer);

The first assertion ensures count is incremented by 1. The expression \old indicates that the count value is the value before calling add(). The second assertion indicates the customer ID is now active. The pre- and post-conditions are specified as:

(1) If the customer ID is already active, the same customer cannot be added;
(2) Increasing the customer count will make the customer ID active.

Violation of either one or both will be considered as illegal and prohibited. If the add() method implementation did not follow JML specification, we would get error debugging feedback like the following:

By reading the debug feedback, we can get the following information:

(1) The application is stopped in an object;
(2) The object is theManager of class CustomerManager;
(3) The error occurred when calling the add() method;
(4) The error is a violation of a precondition of add();
(5) The violation is idActive;
(6) The BasicCustomerDetails object (theCustomer) caused the error when passed as an argument;
(7) The call sequence causes the problems: The changeCustomer method of the CustomerManagerUif class (Customer Manager user interface) calls the add() method of the CustomerManager class;

In summary, we conclude that the CustomerManagerUif class's changeCustomer method is the problem: it is trying to add() an activated ID of the BasicCustomerDetails object, which is illegal.

In supporting the design by contract without slowing down the program execution, the contract testing can be manipulated by turning it on or off according to customer need.

III. MARK-AND-SWEEP GARBAGE COLLECTION

A. Garbage Selection Algorithm

The mark-and-sweep algorithm is based on tracing through the working memory, which includes a mark phase and sweep phase. In the mark phase, the collector does a tree traversal of the entire "root set", marking all reachable objects, while the remaining memory cells are unreachable. During the sweep phase, unreachable objects are returned to the free list. The most notable disadvantage is that the entire system must be suspended during collection, also known as a stop-the-world event. In order to avoid this halting interruption, we adopted an interleaved garbage collector and user program which is called incremental collection. In our approach for JML specification, we also adopted tri-color marking, which divides the heap node into black, gray, and white sets. The tri-color method can be performed "on-the-fly", without halting the system for significant time periods.

(1) The black set is the set of reachable objects that the garbage collector has visited and all their referenced objects.
(2) The gray set is the set of reachable objects the garbage collector has not visited; or, visited but not all their referenced objects; or, the reference relationship has been changed by the user program.
(3) The white set is the set of unreachable objects the garbage collector has not yet visited. At the end of the tracking phase, these are the garbage.

The garbage collection process is divided into mark, sweep, and idle phases. During the mark phase, each object in memory has a flag (a single bit) reserved for garbage collection and a stack data structure to achieve the tri-color abstraction: 1) a marked object not in the stack is considered black; 2) a marked object in the stack is considered gray; 3) an unmarked object not in the stack is white. Although additional data structures needed for the mark phase will increase the memory space required for the collector, they will also shorten the time used for marking survived objects in the stack. If the stack is not empty, every time a gray object is visited, the garbage collector will mark all the non-black objects that are referenced by the current object to grey, and mark the current object to black. This process will continue until the number of visited objects meet the thresh value, and go to the sweep phase. In the sweep phase, not only the white objects are recycled, but also all the black objects are marked white for the next round before entering the idle phase. Once the empty space in the stack is less than the thresh value, a new mark phase is started again.

Copyright (c) IARIA, 2015. ISBN: 978-1-61208-389-6

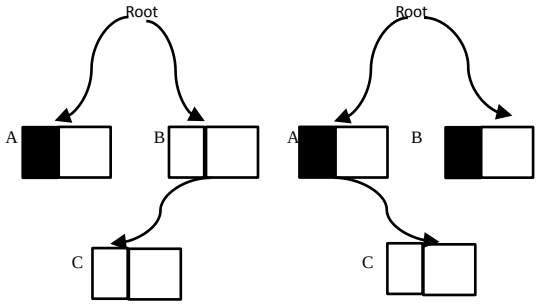

Figure 2. User programs violate garbage collection.

B. *The Write Barrier*

The role of the write barrier is to prevent error caused by a changed reference graph from the user program. Figure 2 shows the user program has changed pointer A to point to object C, and we do not know if there are other references to object B. If the objects B and C at the end of the mark phase are still white, then the garbage collector must ensure they are marked, otherwise they will be treated as garbage in the sweep phase. In this case, an object needs to be protected by a write barrier. Otherwise, there must be an object that is still reachable by the user program but is marked white. Thus, two conditions must be met at marking phase:

(1) A reference to a white object is written to a black parent object, and this is the only reference to this white object.

(2) The original reference to the white object is eliminated.

An object is retained if one or both of the conditions failed. We used Dijkstra's algorithm for color updating. Every time a reference to a white object is created, regardless of the color of its parent, this white object is marked gray. When the collector traverses the heap, there will be no reference from a black object to a white object. For a reachable white object, there must be a path from gray to white. When the collector traverses the stack, condition 1 will fail. This is the solution for the communication between the user program and the collector.

IV. GARBAGE COLLECTOR JML SPECIFICATION

The JML specification for the garbage collector is discussed in this section. Assume the *pointerSet* is the memory set for the garbage collector, including white, gray, black, and free sets. The *freeList* is a linked list used to store the recycled idle objects. The *stack* together with the flag bit is used for marking the objects. The detailed JML specification of the garbage collector is as follows:

```
/*@     require p.getAddr()>=ST&&p.getAddr()<=ED;
  @     assignable p.color,stack;
  @     ensures    p.getColor()==Color.BLACK;
  @                stack.peek()==p;
  @              stack.count==\old(stack.count+1);
  @ also
  @     requires   p.getAddr()<ST||&p.getAddr()>ED;
  @     assignable \nothing;
  @     signals_only IllegalArgumentException;
  @*/
public void markField( /*@non_null @*/ Pointer p);
```

Figure 3 JML specification for the markField.

The *markField* function has a constraint that the *non_null* parameter passed is not an empty pointer. If the pointer address is not within the address space managed by the collector, *assignable \nothing* cannot modify the stack, otherwise it is painted gray. According to the coloring, set *p* flag and push to the stack. The function returns the top element, *stack.peek()* which is *p*, and the number of elements in the stack increases by 1, which is *\old(stack.count)* plus 1.

```
/*@requires phase==Phase.MARK&&stack.count==0;
  @   assignable Phase;
  @   ensures phase==Phase.MARK;
  @             \not_modifed(stack);
  @             \not_modifed(pointSet);
  @             \not_modifed(freeList);
  @also
  @   requires phase==Phase.MARK&&stack.count>0;
  @             (\forall Pointer p;
  @                 pointerSet.contains(p)&&p.getcolor==Color.BLACK;
  @                   p.accessible(root));
  @   invariant NumMark>=MARKNUM&&NumMark<=MARKNUM;
  @   assignable Phase,stack,pointSet.color,pointSet.Field;
  @   ensures (numMark==MARKNUM&&phase==Phase.MARK);
  @             (\forall Pointer p;
  @               pointerSet.contains(p)&&p.getcolor()==Color.BlACK;
  @     p.getField1().getcolor()==Color.BlACK&&p.getField1==stack.peek()&&
  @     p.getField2().getcolor()==Color.BlACK&&p.getField1==stack.peek()));
  @*/
  public void mark();
```

Figure 4. JML specification for the mark function.

The precondition of the *mark()* function is that garbage collection is in the mark phase, and the operations are dependent on the status of the stack. When the stack is empty, that means no nodes need to be visited, and no operations need to be done, thus the garbage collection goes to the sweep phase directly. If the stack is not empty, the mark stage is started. The assumption is that all the black and gray objects are reachable by the root. Each time a top element (a black node) is popped, all its referenced objects need to be marked gray, and the number of marked objects increases until reaching the threshold value. In the mark phase, the pop and push operations modify the stack, the flag, and the address space, while other variables that are not declared in the *assignable* remain unmodified. This ensures that the mark process did not modify the user information, idle list, or the current sweep position in the main memory.

```
/*@   invariants numSwept>=0&&numSwept<=SWEEPNUM;
  @   invariants sweepCur>=START&&sweepCur<=END;
  @   requires phase==Phase.SWEEP&&START<=p.getAddr()<=END;
  @   assignable freeList,pointSet.color;
  @   ensures numSwept=SWEEPNUM;
  @         (\forall Pointer p;START<=p.getAddr()<=sweepCur&&
  @           \old (p.getColor()==Color.BLACK);p.getcolor==WHITE)
  @               (\forall (Pointer p;sweepCur<=p.getAddr()<=END&&
  @         \old (p.getColor()==Color.BLACK);p.getcolor==BLACK)
  @                   (\forall (Pointer p;START<=p.getAddr()<=sweepCur&&
  @         \old(p.getColor()==Color.BLACK);p.getcolor==WHITE&&
  @       freelist.getlast()==p&&p.freeField<==>TRUE);
  @also
  @   requires   phase==Phase.SWEEP&&p.getAddr()>=END;
  @   ensures phase==phase.IDLE;
  @*/
public void sweep();
```

Figure 5. JML specification for the sweep function.

Similar to the *mark()* function, the precondition for the *sweep()* function is that garbage collection is in the sweep phase. Its task is to recycle a certain number (*SWEEPNUM*) of garbage. The *invariant* keyword describes, during sweep the *numSwept* (already swept objects) and *sweepCur* (current sweep address) should both be within the valid range. Changing in *\old(p.getColor)* (before call) and *p.getColor* (after call) means all the objects before *sweepCur* are swept. For the objects after *sweepCur*, *p.getColor* remaining unchanged means the objects were not visited. By setting the second identification bit *p.freeField* we can check whether the garbage is in the free list. The assertion *Freelist.getlast () == p* verifies if it is indeed recycled in the free list. Finally, all the visited black objects are marked white for the next round sweep.

Garbage collector calls the corresponding functions according to *phase* status. Whether to start the next round of mark-sweep is based on the number of idle objects (*FREENUM*). The garbage collector constantly monitors the memory to make judgments. When the size of the free list, *freelist.size*, is less than the threshold, the *start_marking* function sets *phase* to mark status. The *sweepCur* starts scanning from low address and marks the root node to mark phase. Its JML specification is as follows:

```
/*@ requires freelist.size()<FREENUM&&phase==Phase.IDLE;
  @ ensures sweepCur==START;
  @              phase=Phase.MARK;
  @            root.color==Color.BLACK;
  @           root==stack.peek();
  @         stack.size=\old (stack.size+1);
  @*/
public void startMarking();
```

Figure 6. JML specification for start_marking().

In order to make the garbage collector and the user program interact properly, the *allocate()* function needs not only to allocate space to the user program from the free list, but also needs to avoid treating objects as garbage in the un-swept address segment. After assigning space to the user program (*\result* is assigned starting address), the length of the free list and the number of the idle objects (*numfree*) will both be reduced. Marking objects after *sweepCur* black can ensure the unprocessed objects are not treated as garbage.

Copyright (c) IARIA, 2015. ISBN: 978-1-61208-389-6

```
/*@ requires freelist.size()>0;
  @ assignable freelist,pointSet.color;
  @ ensures \result=\old(freelist.getFirst());
  @              freelist.size()=\old(freelist.size()-1);
  @              numfree=freelist.size();
  @            (\result.getAddr()<sweepCur;||(\result.getAddr()>=sweepCur&&
  @               \result.getColor==Color.BLACK;)
  @*/
public void allocate ( );
```

Figure7. JML specification for allocate function

The *assignable* constraint in the write barrier only changes the color of the node, while data in memory is not changed before or after the execution of write barrier. As long as there is a pointer from black to white in the stack, the white are marked gray to ensure the user program will not interfere with the execution of the garbage collector.

```
/*@ requires phase=Phase.MARK;
  @ assignable stack,pointSet.color;
  @ ensures (\exist Iterator it; PointerSet.iterator()>=START/(sizeof)(Pointer)
  @          &&PointerSet.iterator()<=END/(sizeof)(Pointer)&&
  @ \old(it.getcolor==Color.BLACK)&&\old(it.next().getColor==Color.WHITE);
  @     it.next().getColor==Color.BLACK&&stack.count=\old(stack.count+1)&&
  @        stack.peek()==it.next());
  *@/
  public void djikstraStroe(Pointer field,Pointer val);
```

Figure 8. JML specification for djikstraStore().

The examples shown above illustrate that the JML specification can efficiently specify the pre- and postconditions for the mark, sweep, and write barriers of the garbage collector. Based on the system requirements, the JML specification can be applied to the entire garbage collector to improve the correctness and reliability.

V. CONCLUSION AND FUTURE WORK

We discussed the JML specification for the interaction between the garbage collector and the user program. The assertion is based on DBC pre- and postconditions in Hoare-style logic. This study focuses on real environment memory objects without abstraction, which is more reliable to some extent. The JML specification covers both normal and abnormal behavior which can accurately describe the real-time environment. Runtime execution of the assertions is more suitable for object-oriented software. In our approach, program execution is not only the result of a function generation process, but also an assertion checking process. This approach can improve correctness and reliability for the garbage collector, quickly position errors, and handle abnormal behavior during collection. For the future, we will focus more on DBC implementation in JML, improve accuracy for describing various types of garbage collectors, and their implementation on generational concurrent garbage collectors.

REFERENCES

[1] L. Birkedal, N. Torp-Smith, and J. Reynolds, “Local reasoning about a copying garbage collector,” Proc. 31st ACM Symp On Principles of Prog .Lang. pp 220-231, 20014.

[2] D. Russinoff, “A mechanically verified incremental garbage collector,” Formal Aspects of Computing, vol. 6, pp 359-390, 1994.

[3] L Chun-xiao, Y. Chen and Li Long, “Garbage collector verification for proof-carrying code,” Journal of Computer Science and Technology vol 22, pp. 426-437, 2007.

[4] M. Ben-Ari, “Algorithms for on-the-fly garbage collection,” ACM Transactions of Principles on Programming Languages and Systems, vol. 6, pp. 333-344, 1984.

[5] G. Leavens, A. Baker, and C. Ruby, “Preliminary Design of JML: A Behavioral Interface Specification Language for Java,” ACM SIGSOFT Software Engineering Notes, vol 31, pp. 1-38, 1999.

[6] G. Leavens, A. Baker and C. Ruby, “JML: A Notation for Detailed Design.,” in Behavioral Specifications for Businesses and Systems, Chapter 12, H Kilov, B Rumpe and W Harvey, Eds. Kluwer, pp. 175-188, 1999.

[7] G. Leavens and Y. Cheon, “Design by Contract with JML,” [Online]. Available from:. http://www.eecs.ucf.edu/~leavens/JML/jmldbc.pdf. Accessed 3/12/2014.

[8] G. Leavens, E Poll, C. Clifton, Y. Cheon, C Ruby, D Cok, J. Kiniry, P. Chalin, D. Zimmerman and W. Dietl. “JML Reference Manual,” (DRAFT), [Online]. Available from: http://www.jmlspecs.org/OldReleases/jmlrefman.pdf. Accessed 3/12/2014.

[9] L. Burdy, Y. Cheon, D. Cok, M. Ernst, J. Kiniry, G. Leavens, K. Rustan M. Leino and E Poll, “An overview of JML tools and applications,” International Journal on Software Tools for Technology Transfer, vol. 7, pp. 212-232, 2005.

[10] R. Mitchell, J. McKim and B. Meyer, Design by contract, by example, Redwood Citiy, Addison Wesley, 2002.

Copyright (c) IARIA, 2015. ISBN: 978-1-61208-389-6