Specifying Multi-Threaded Java Programs

A Comparison between JML and Separation Logic

Ronald Burgman
Faculty of Electrical Engineering, Mathematics and Computer Science
University of Twente
r.w.burgman@student.utwente.nl

ABSTRACT

Recently Hurlin designed an extension to separation logic, that made it possible to specify multi-threaded Java-like programs with fork/join constructs and reentrant locks. In this paper we aim to evaluate the usability of Hurlin's proposed method. This is done by comparing Hurlin's variant of separation logic with JML for the specification of a well-known multi-threaded design pattern.

Categories and Subject Descriptors

F.3.1 [**Logics and Meanings of Programs**]: Specifying and Verifying and Reasoning about Programs—*Specification techniques*

General Terms

Experimentation, Verification

Keywords

Program specification evaluation, separation logic, JML, multi-threading

1. INTRODUCTION

It is generally acknowledged that complex computer software is often subject to errors, also known as bugs. The damage caused by these bugs can be severe. Therefore, the elimination of bugs from computer software has, since the beginning of computer science, been an important topic. Many solutions for this problem have already been proposed, using a wide variety of techniques.

At the basis of most of the proposed solutions stands the principle of program specification. A program specification contains a description of the program's behavior and without such a definition it is difficult to decide whether the program behaves correctly or incorrectly. Modern approaches to make such specifications often use formal mathematical-based languages, called model languages, to describe or model the behavior of computer software. The mathematical nature of model languages makes them precise, which makes it easy to automate the validation process. In this paper two such modeling languages will be used, namely the Java Modeling Language and separation logic.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
12th Twente Student Conference on IT January 22, 2010, Enschede, The Netherlands.
Copyright 2010, University of Twente, Faculty of Electrical Engineering, Mathematics and Computer Science.

1.1 Separation logic

Separation logic was introduced by Reynolds [4]. It is designed as an extension of Hoare logic, a special logic created to reason about software properties. Separation logic distinguishes itself from other logics by the reference to a stack and a heap, which together represent the memory being used by the specified program. This explicit way of referring to the memory gives the opportunity to reason about the contents of the memory and about the relation between parts of the memory.

Each thread has a unique stack, which is the basic stack associated with any program execution. The heap, however, is the same for all threads and contains the data that is accessed through pointers and therefore data that can be shared between threads. Separation logic also introduces some operations to manipulate and compare the heap. Specifically these operations allow the heap to be split in disjoint parts at the specification level. This makes it possible to determine the parts of the heap that are shared with other objects and thus which pointers are aliases and which are not.

The separation logic that is used in this paper is described in Hurlin's recent PhD thesis [3]. In his thesis Hurlin describes a variant of separation logic that is able to specify multi-threaded Java-like programs with fork/join constructs and reentrant locks.

1.2 Java Modeling Language

This research also makes use of another modeling language besides separation logic, namely: Java Modeling Language (JML). JML is, like separation logic, based on Hoare logic and is able to make specifications of Java programs. JML aims to be complete, formal, but understandable by the average Java programmer. The basis of JML specifications are assertions, which can be placed in the Java documentation or a separate specification file. This approach makes JML suitable for any given Java compiler. Also, a lot of validation tools for JML are available [1], automating program validation.

JML is, in contrast to the separation logic variant used, not suitable for multi-threaded programs, but JML is more mature and also some practical examples of the use of JML

are available, therefore it will be used in this research as an example and as a reference.

1.3 Related work

Reynolds [4] introduced separation logic. In his article he describes the basic concepts of separation logic. The separation logic we will use is an extension of this separation logic.

Hurlin [3] describes a way to use separation logic for specifying modern multi-threaded Java-like programs with fork/join constructs and reentrant locks. The thesis focuses a lot on the soundness of the concept and little on the practical use of the proposed method.

1.4 Problem Statement

In this research the focus is on the method described by Hurlin. His adaption of separation logic for multi-threaded programs took separation logic one step closer to an application in the 'real world'. But before that goal can be realized, first the practical characteristics of separation logic need to be examined. In his thesis Hurlin focuses on proving the soundness of his theory, but spends little time on the usability of his method. This resulted in a relatively complex method of which it is uncertain if it is usable in a practical situation.

Determining the usability of Hurlin's method is done by using his method for the specification of a well-known multi-threaded software pattern. Because it will be very difficult to objectively determine the usability, only the relative usability of the proposed method will be determined. This is done by comparing the specification in separation logic to a specification in Java Modeling Language (JML).

2. RESEARCH APPROACH

2.1 Research Questions

The main research question of this research is:

- What is the usability of the method for program specification with separation logic described by Hurlin [3] when compared to JML?

This research question contains two issues, which need to be addressed before the actual research:

- What is the suitability of JML as a reference language?
- How is (relative) usability defined?

2.2 Java Modeling Language

Because Java Modeling Language (JML) is a well-developed specification language for Java programs, which is used in practical situations [1] it can be said that JML has a good usability and therefore it is a good choice as a reference for comparing the usability of other specification languages. Another advantage is the prior knowledge of JML with the researcher, which allows to save a significant amount of time that is scarce for this project.

Unfortunately JML also has a disadvantage, which is the poor support for multi-threaded programs. There has been at least one attempt [5] to add multi-threaded functionality to JML, but this research has not yet advanced enough to be included in JML for the purpose of this research. Because the specification method designed by Hurlin is specifically for multi-threaded programs, this is a significant deficiency.

However because of the time constraints on the project, it will not be feasible to replace JML with another, better suited, specification language.

2.3 Measurement

As stated earlier it is difficult to objectively determine the usability of the proposed method and therefore the proposed method will be compared to JML. Both separation logic and JML are used to specify the behavior of some well-known design pattern for multi-threaded programs. Both specifications are then compared on the following criteria.

- The time it took to make the specification.
- The length of the specification.
- The completeness of the specification

In this case, completeness refers to the precision of the specification. Because of the differences between separation logic and JML, situations will arise in which one of the languages will be more suitable to specify a specific part of the code, resulting in different contracts. One of these contracts will specify fewer characteristics and thus is less complete, which is a disadvantage. A specification is basically a list of properties and when this list becomes longer the specification is obviously more complete. Therefore counting the number of properties in both specifications will be a good way to measure the completeness of both specification languages. Because JML is unsuited to describe multi-threaded programs, it is expected that the completeness of JML will be lower than the completeness of separation logic.

It is also likely that the completeness will have an influence on the other criteria. It is, for example, intuitive to state that a specification, which is more complete, takes a longer time to make and also will be longer than a specification which is less complete. This should be kept in mind when comparing the results.

3. THE ALGORITHM

The test specification is written for the mergesort algorithm. This is a well-known divide and conquer type algorithm and it is therefore easy to create a multi-threaded version of the algorithm. Also, with a few modifications, as described below, it will contain all the required properties we need for this example. The complete original and modified versions of the algorithm can be found online [2].

Because JML will probably perform well in a single-threaded environment and separation logic will probably perform well in a multi-threaded environment it will not be sufficient to only specify the multi-threaded version of the algorithm. Therefore we will also make a specification of the single-threaded version of the algorithm.

3.1 Linked List

Both the multi-threaded and the single-threaded version of the algorithm make use of a linked list, so the algorithm can retain some efficiency while rearranging the elements.

The nicety of rearranging the elements instead of copying them is discussed in the next section. The general structure of the linked list class is found in Appendix B as Code 3

The implementation of the single-threaded and multi-threaded version of the linked list does not differ except for one point. In the multi-threaded version every element has its own lock. All operations on a single element require the lock of that element to be acquired before the operation, therefore all methods will start whit *this.lock* and end with *this.unlock*. The only exception to this is the *addAll* method, which looks as follows:

```
public void addAll (LinkedList l)
  if (next == null)
    next = l;
    this.unlock();
  else
    next.lock();
    this.unlock();
    next.addAll(l);
```

As can be seen the *addAll* method never locks the 'this' element, but only the next element. This allows for lock coupling, which means the next element is locked before the current element is unlocked. This way of locking guarantees that while executing the *addAll* method there will never be a moment where not a single element is locked. This, in turn, will guarantee that no two calls to the *addAll* method are able to overtake each other.

Lock coupling, in this example, makes sure that elements, which are added first, will actually be appended first at the end of the list. This is not a necessary property in this example, but lock coupling is part of Hurlin's thesis and it is added to be able to test the usability of all the properties described in the thesis.

3.2 Multi-threaded Mergesort

For the merge sort algorithm to be interesting to test Hurlin's method, it needs to contain the basic structures which were the focus of Hurlin's thesis. This means it must be multi-threaded and needs to contain some shared resource, which can be locked. This shared resource will be the list, which is to be sorted. As a consequence the *merge* method will no longer be allowed to copy the list, instead it rearranges the elements. To keep the algorithm as efficient as possible we will explicitly make use of a linked list. A linked list does not require element shifting as will be required for an array-based structure.

The algorithm is made multi-threaded by replacing the recursive calls in the *sort* method by a fork/join pair. The new threads will receive two parameters, namely the first element of the linked list to be sorted and the number of elements to be sorted. This second parameter is necessary because we can no longer cut the list in smaller pieces and therefore can no longer derive the number of arguments to be sorted from the size of the list. The code of the *sort* method with these modifications can be found in Appendix B as Code 1.

As said above, the *merge* method of the algorithm is modified in such a way that the elements of the linked list are rearranged instead of copied to a new list. To efficiently do this, a reference to the last element in the merged list is stored, so we can easily add extra element(s) at the end of the list. The pseudo-code of the merge method can be found in Appendix B as Code 2.

3.3 Single-threaded Mergesort

The single-threaded version is made to resemble the modified multi-threaded version of the algorithm as much as possible, instead of the original single-threaded version. This makes it easier to compare the multi-threaded and single-threaded versions. Because the multi-threaded version is used as a basis, the biggest difference between the single-threaded and multi-threaded version is that fork/joins in the multi-threaded *sort* method are replaced by recursive calls in the single-threaded version.

4. SPECIFICATIONS

This section discusses the specifications. Because of size constraints the complete specifications are omitted from this paper, however the most important and interesting parts are presented for discussion. The complete specifications and separation logic proofs can be found online [2]. The JML specifications used in this paper are validated with the ESC/Java [6] tool and the separation logic specifications are manually proven correct.

4.1 Single-Threaded Specifications

4.1.1 JML

For the contract of the linked list we introduce a model method *isSorted*, a model variable *size* and an invariant. The model method *isSorted* is used to specify that a list is sorted. This makes the contracts of the methods that work with sorted lists considerably more readable. The model variable *size* makes it possible to refer to the size of the list, without referring to the implementation of the list. We also introduce an invariant that specifies that the *size* is at least one. In this context its meaning is not particularly useful, but because we want to compare the ability of JML and separation logic to make complete contracts it is still added. This part of the specification is found in Appendix C as Specification 4.

Most of the contracts of the linked list methods are trivial and generally they look as follows.

```
/*@
  @ ensures this.getNext() == next;
  @*/
public void setNext(LinkedList next)
```

The only contract that is more complicated is the contract of the *addAll* method, which can be found in Appendix C as Specification 5.

The contract of *addAll* has three parts. The first simply specifies that the list size increases with the size of the added list. The second part is a bit more difficult to read, but specifies that the elements which were in the original list are not altered. Finally, part three ensures that the new elements are added at the end of the list, but are not further edited.

The contract of the *sort* method in the merge sort algorithm contains three parts and looks as follows:

```
/*@
  @ (1) requires list != null;
  @ (2) ensures list.isSorted();
```

```
  @ (3) ensures list.get(size-1).getNext() ==
      null;
  @*/
public LinkedList sort(LinkedList list, int
    size)
```

The sort method (1) requires that the list parameter is not null, meaning there actually is a list to be sorted. In turn it (2) ensures that the list is sorted and (3) that the last element of the sorted list has a null reference to the next element. The last property is a requirement of the *merge* method. This method assumes that the lists, which need to be merged, end with a null reference. An alternative would be to explicitly pass the size of the sorted elements to the *merge* method, then the list may end without a null reference. One of these two properties is necessary, because it is possible for loops to exist in the list as a result of the rearranging of the elements and without explicitly declaring the end of the list the algorithm can get stuck in an endless loop.

As said in the last paragraph the *merge* method requires that the list in its parameters end with a null pointer, furthermore it requires that the parameters are sorted and ensures that the result again ends with a null pointer and is sorted. The contract looks as follows:

```
/*@
  @ requires l1.sorted &&
      l1.get(l1.size-1).getNext() == null;
  @ requires l2.sorted &&
      l2.get(l2.size-1).getNext() == null;
  @ requires \result.sorted &&
      \result.get(\result.size-1).getNext() ==
      null;
  @*/
```

4.1.2 Separation Logic

The JML contracts above are also specified in separation logic to be able to compare the contracts. Some properties turned out to be easier to specify in separation logic, while others were considerably more difficult. In this section we will show some examples of both.

A good example property is the *sorted* predicate. This predicate is used as the counter part to the *isSorted* model method in the JML specification and is able to show the differences between JML and separation logic clearly. The property looks as follows.

```
public pred sorted = (ex Perm p)
   (fa LinkedList l)
   PointsTo(next, p, l) -*
   (this.value <= l.value * l.sorted);
```

The predicate consists of two parts separated by a magic wand. The first part in front of the magic wand specifies a possible next element. When this next element exists, the second part specifies that the value in the current element is smaller than or equal to the value in the next element, meaning these two elements are sorted. Then a recursive call is made to the next element, specifying the *sorted* property for the rest of the list.

The *sorted* predicate contains some of both the advantages and disadvantages. Because in separation logic it is easy to reason about explicit memory structures, such as linked lists, it is easy to specify properties about such a structure. The result is a *sorted* predicate that is easier to read than the model method that is the counter part in the JML specification. A disadvantage is that often variables have to be explicitly quantified. While in this example the issue is not very important because we do not use many variables, it greatly reduces readability when more variables are needed, while it adds little or none to the meaning of the predicate. A solution could be to implicitly quantify the unknown variables.

The *sorted* predicate can also be used to show some weak points in separation logic, for example when the predicate needs to be defined for an array. An array with five elements is modelled in separation logic as a list of adjacent cells. According to Reynolds [4] it looks like this.

$$\text{array } a \mapsto a_1, a_2, a_3, a_4, a_5;$$

In this case the array definition is fairly simple, but only because it is short and we know the length. If the array would be much longer or would have an unknown size, this will no longer be a usable way to define an array.

Besides the simple definition of the array, it is also quite hard to make statements about the array because of the lack of indexing. Indexing would make it possible to specify a list is sorted in one statement instead of a separate statement for every adjacent pair in the array.

Both problems are absent in JML, in which memory structures do not have to be defined and where elements can be quantified with a universal or an existential quantifier. Although both quantifiers are available in separation logic they cannot be used in this context. Adding some syntax so the above statements can be described with the use of quantifiers will make them more flexible and easier to read. But because separation logic always needs a reference to the memory, a specification in separation logic will always be more complicated than a similar specification in JML.

4.2 Multi-Threaded Specifications

4.2.1 JML

It is impossible to make a specification in standard JML for a multi-threaded program. A large part of a JML specification consists of constraints on variable contents through pre- and post conditions. As long as these variables are not locked, which is generally the case at the start and the end of a method call, the content of these variables can be modified by other threads making pre- or post conditions invalid.

4.2.2 Separation Logic

As said earlier in the algorithm section: the multi-threaded version of the algorithm uses a linked list of which al individual elements can be locked separately. Separation logic implements a lock by associating a lock with a resource invariant. When you acquire the lock you also acquire the invariant and visa versa. In our algorithm if we acquire a lock on an element, we also want to have complete access to the object. This means that the lock invariant must contain write permissions to all the fields of a linked list element. In separation logic the invariant looks as follows.

```
pred inv = Perm(value, 1) *
```

```
(ex List L) PointsTo(next, 1, L);
```

This invariant contains all the permissions we need, namely write permission to the value in this element and write permission for the reference to the next element.

It is the task of the *constructor* to initialize the element and guarantee the lock invariant. This is made explicit in the contract of the *constructor* by adding an *initialized* post condition and the execution of the *commit* instruction. The *constructor* and the proof of its contract are added in Appendix section D as Specification 6 and as Proof 7.

Most other contracts are simple; they require that the element is unlocked but lockable and that the lockset at the method's end equals the lockset at the method's start. In separation logic this contract is formalized as follows.

```
/*
 * requires LockSet(S) * ! (S contains this) *
    this.initialized;
 * ensures LockSet(S) * ! (S contains this);
 */
```

The only exception to this is again the contract of the *addAll* method, which is more complicated because of the lock coupling within this method. This is expressed in the contract as follows:

```
/*
 * requires LockSet(this . S) * inv *
     next.initialized;
 * ensures LockSet(S) * !(S contains this);
 */
public void addAll(LinkedList l)
```

The use of lock coupling requires that this element is already locked when the *addAll* method is called. This can be seen in the contract in the *LockSet(this . S)* part, meaning the lockset contains the locks of *this* and some unknown set of objects *S*. Because this element is locked, the lock invariant is also required and because we intent to lock the next element it needs to be lockable: *next.initialized*. Although at first glance this contract looks fine, it lacks one important property, namely that the next element is currently unlocked. This can be easily seen in the proof of the addAll method, which is added in Appendix section D as Proof 8.

When we want to lock the next element it is not known if it is already locked by this thread or not, so the method might fail. In his thesis Hurlin [3] solves this problem by introducing the property *traversable*. This property uses a class parameter to specify an *owner* object, which owns this element. The *traversable* property uses this parameter to specify that no element owned by a certain object or thread is in the locksetof the current thread. The *traversable* property can then be used to prove that the call to the *lock* method will succeed.

However the *traversable* property cannot be used in this way in our specification. This is because of the way ownership is specified, through a class parameter, which makes the ownership static. Since our list is passed between different threads, the ownership needs to be more dynamic and needs to be able to be altered. Although it is beyond the scope of this paper to find a solution for this problem, we still considered a few possible solutions: to allow reassignment of the ownership, to specify a predicate and to adapt the contract of the *addAll* method.

The first attempt was simply to allow the ownership to be altered. This is however still not dynamic enough. Ownership is past between elements in a recursive manner, which would mean that a change of ownership would require the whole list to change ownership. Because the list is split between different threads, which work on this list simultaneously, different parts of the list need to have different ownerships. Therefore it is not enough to simply allow a reassignment of the ownership, but a different way of defining ownership for multiple elements at once is also necessary.

As a second attempt we tried to specify a recursive predicate which defined the *traversable* property, but this is not possible. Any such property requires read permissions for every element in the list, or at least for each element for which the ownership will be defined by this predicate. These permissions are however associated with the *lock invariant* and can never be acquired without the lock. So to specify that an element is unlocked, first the lock needs to be acquired, after which the element is obviously not locked anymore.

As a last attempt we tried to add the condition *!(S contains next)* to the requirements. In this specific case it is allowed because we already acquired the lock invariant of this element, so we have permissions to reference the next element. This solves the problem with the call to the *next.lock* method, but creates another one. When the *next.addAll* method is called, its precondition requires that the *next.next* object is unlocked, which is unknown at this point in execution.

Although multiple attempts have been made, a simple solution cannot be found. Therefore more research is needed to develop a way to assign ownership more dynamically.

5. COMPARISON OF JML AND SEPARATION LOGIC

As said, JML and separation logic are compared on three properties: completeness, the time it took to make a specification and the length of the specifications.

The number of properties can be found in Table 1 and they are split in the following categories: class, JML or separation logic (SL) and single-threaded (SL) or multi-threaded (MT). In this count three kinds of properties are taken into account: preconditions, postconditions and invariants.

The predicates in separation logic and the model declarations in JML are not counted as properties, because in their own they do not have a meaning. Instead they are counted if they are used in an invariant or a pre-/postcondition. If a property occurs in multiple contracts it is counted multiple times. In *\forall* quantifiers in JML we do not see the first two parts of the statement, which specify the range of the quantifier, as a property, but only the third part.

As can be seen, the difference between the numbers of the single-threaded version of the algorithm is small. The cause of this difference is the absence of the *\old* and the *pure* keywords in separation logic. Therefore properties, which make use of these keywords, cannot be defined in

separation logic. The reason for the difference in numbers for the multi-threaded version of the algorithm is as discussed in the previous section; it is impossible to describe any JML property in the multi-threaded version of the algorithm, because the way JML specifies variable content does not work in a multi-threaded environment. This clearly shows that this variation of separation logic was designed for multi-threaded programming and JML was not.

In Table 2 the time taken to make the specifications is shown. The difference in the time taken is actually a bit larger, because of some problems with the ESC/Java tool, which did not validate the specifications correctly. The time taken to solve the problems with the tool are partially included in this count, but the exact amount of time spend on this problem is unknown.

The difference in time is not big, but in favor of JML. Part of this difference is related to the inexperience of the researcher with separation logic, but it probably still is possible to say that it is easier to make a specification in JML than it is in separation logic.

Finally the length of the specifications is measured. For this we count the number of characters in the specification excluding whitespace. The results are as shown in Table 3. Of cource no numbers are available for the multi-threaded specification in JML. The last big difference in the table is between the single-threaded specifications of the linked list class. But as is seen in Table 1 this specification is not complete in separation logic, which explains the difference.

6. CONCLUSION

JML and separation logic are two specification languages, which rely on two completely different models. JML is used to specify the contents of variables and return values and because the input and output of a program rely on these variables, JML can also specify a program's behavior. In a multi-threaded environment, however, processes can be arbitrary interleaved and it becomes difficult, or even impossible to make claims about variable contents, which completely nullifies JML's ability to make program specifications.

Separation logic, in contrast to JML, does not try to specify input and output, but focuses on actions and permissions to perform these actions. Therefore separation logic is still able to make claims in a multi-threaded environment where the content of variables cannot always be guaranteed.

While separation logic is specially designed to work in a multi-threaded environment, it can also be used together with or instead of JML in a single-threaded environment. But the abilities of separation logic are less compared to JML. Separation logic uses a more complicated model than JML and therefore the use of separation logic will result in a more complicated specification. The support to describe variables on a high level, like with quantifiers in JML, is also absent.

Finally it can be concluded that separation logic and JML both have their advantages and disadvantages and that they mainly support each other, but not replace each other.

7. FUTURE WORK

Throughout this paper two issues have been identified, which still need more research: the traversable property and ownership and the use of arrays. The traversable property is a good solution to specify a certain set of objects is unlocked, but it cannot be used yet in every situation because the ownership, which is used by the traversable property, is only assigned at object creation and can no longer be altered at a later time. Therefore a different way of dynamically assigning this ownership is needed. Also a different way of assigning the ownership of multiple objects at the same time is needed. This is now done through recursion, which for example does not allow for elements of the same list to have different owners.

The second issue is the support for arrays, which is still poor. To easily work with arrays a better syntax is needed to define arrays and indexing is needed to easily make statements about arrays.

8. REFERENCES

[1] L. Burdy, Y. Cheon, D. Cok, M. Ernst, J. Kiniry, G. Leavens, K. Leino, and E. Poll. An overview of JML tools and applications. *International Journal on Software Tools for Technology Transfer (STTT)*, 7(3):212–232, dec 2005.

[2] R. Burgman. Specifications and code. http://purl.org/burgman/separation-logic.

[3] C. Hurlin. *Specification and Verification of Multithreaded Object-Oriented Programs with Separation Logic*. PhD thesis, l'Université de Nice-Sophia Antipolis, sep 2009.

[4] J. Reynolds. Separation logic: A logic for shared mutable data structures. In *Logic in Computer Science, 2002. Proceedings. 17th Annual IEEE Symposium on*, pages 55–74, 2002.

[5] E. Rodríguez, M. Dwyer, C. Flanagan, J. Hatcliff, G. Leavens, and Robby. *ECOOP 2005 - Object-Oriented Programming*, volume 3586/2005, chapter Extending JML for Modular Specification and Verification of Multi-threaded Programs, pages 551–576. Springer Berlin / Heidelberg, 2005.

[6] K. Software. ESC/Java2, version 2.0.5. http://secure.ucd.ie/products/opensource/ESCJava2/.

APPENDIX

A. TABLES

	JML		SL	
	ST	MT	ST	MT
Mergsort	9	0	9	34
LinkedList	19	0	11	9

Table 1: No. of Properties

JML	74
Separation logic	80

Table 2: Time taken in min.

	JML		SL	
	ST	MT	ST	MT
Mergsort	300	0	264	203
LinkedList	763	0	521	614

Table 3: Specification Length in characters

B. CODE

```
public void sort()
  if (list.size > 1)
    mid = list.size / 2;
    thread1 = new MergeSort(list, mid);
    thread2 = new MergeSort(list.get(mid),
        size-mid);
    thread1.fork(); thread2.fork();
    thread1.join(); thread2.join();
    list = merge(thread1.list, thread2.list);
  else
    //already sorted
```

Code 1: MergeSort, Sort Method

```
public List merge (List l1, List l2)
  if (l1 < l2)
    result = l1;
    l1 = l1.next;
  else
    result = l2;
    l2 = l2.next;
  last = result;
  while (l1 != null AND l2 != null)
    if (l1 < l2)
      last.next = l1;
      l1 = l1.next;
    else
      last.next = l2;
      l2 = l2.next;
    last = last.next;
  if (l1 != null)
    last.next = l1;
  else
    last.next = l2;
  return result;
```

Code 2: MergeSort, Merge Method

```
public class LinkedList
  int value;
  LinkedList next;

  public LinkedList(LinkedList next, int value);
  public void setNext(LinkedList next);
  public LinkedList get(int i);
  public int size();
  // Adds l at the end of the list
  public void addAll(LinkedList l);
```

Code 3: LinkedList

C. JML SPECIFICATIONS

```
/*@
  @ ensures (\result == true) <==> (next ==
      null) || (this.getValue() <=
      this.getNext().getValue() &&
      next.isSorted());
  @ public model boolean isSorted();
  @*/

//@ public model int size;
//@ private represents size = (next == null) ?
    1 : 1 + next.size;
//@ invariant size >= 1;
```

Specification 4: JML spec, LinkedList

```
/*@
  @ (1) ensures size == \old(size) + l.size;
  @ (2) ensures (\forall int j; 0 < j &&
            j < \old(size);
            this.get(i) == \old(this.get(i)));
  @ (3) ensures (\forall int i; 0 < i &&
            i < l.size;
            this.get(i+\old(size))
            == l.get(i));
  @*/
    public void addAll(LinkedList l)
```

Specification 5: LinkedList, addAll Method

D. SEPARATION LOGIC SPECIFICATIONS AND PROOFS

```
/*
 * requires init * fresh;
 * ensures initialized
 */
public LinkedList (LinkedList next, int value){
  this.next = next;
  this.value = value;
  this.commit();
}
```

Specification 6: LinkedList, Constructor Contract

```
{init * fresh}
public LinkedList (LinkedList next, int value){
  {PointsTo(this.next, 1, null) *
      Perm(this.value, 1) * fresh}
  this.next = next;
  {PointsTo(this.next, 1, next) *
      Perm(this.value, 1) * fresh}
  this.value = value;
  {PointsTo(this.next, 1, next) *
      Perm(this.value, 1) * fresh}
  {inv * fresh}
  this.commit();
}
{initialized}
```

Proof 7: LinkedList, Constructor Proof

```
/*
 * requires LockSet(this . S) * inv * l.initialized;
 * ensures LockSet(S) * !(S contains this);
 */
public void addAll(LinkedList l){
  {LockSet(this.S) * inv * l.initialized}
  {LockSet(this.S) * (ex LinkedList n) PointsTo(next, 1, n) *
  if (next == null){
    //Case 1: n == null
    {LockSet(this.S) * (ex LinkedList n) PointsTo(next, 1, n)
    next = l;
    {LockSet(this.S) * PointsTo(next, 1, l) * Perm(value, 1)
    {LockSet(this.S) * inv}
    this.unlock();
    {LockSet(S) * !(S contains this)}
  } else {
    //Case 2: n != null
    {LockSet(this.S) * (ex LinkedList n) PointsTo(next, 1, n)
    (Open/Close){LockSet(this.S) * inv * l.initialized}
    next.lock();
    {LockSet(this.S.next) * inv * next.inv * l.initialized}
    this.unlock();
    {LockSet(S.next) * !(S contains this) * next.inv * l.init
    next.addAll(l);
    {LockSet(S) * !(S contains this)}
  }
  {LockSet(S) * !(S contains this)}
}
```

Proof 8: LinkedList, addAll method