Specifying Concurrent Programs: a case study in JML and Separation Logic

Martijn Roo
University of Twente
The Netherlands
m.j.roo@student.utwente.nl

ABSTRACT

This paper presents a case study for the verification of concurrent programs. A model for a central printer server was designed, implemented and annotated with a formal specification in JML, extended with syntax for permission-based separation logic. The specification is compatible with the VerCors toolset which is currently being developed at the University of Twente. The goal of this research has been to design and implement a shared data structure with a formal specification that can be used to test future concurrent program verifiers. The correctness of the program is discussed in natural language and an outline for a formal proof is given

Keywords

Program verification; Concurrency; Case study; VerCors

1. INTRODUCTION

Processors get increasingly more cores and software developers try to benefit from this by making more use of concurrency in their programs. With the increase in use of concurrency in software, the demand for verification of concurrent programs increases. Most current verification tools focus on verifying sequential programs and they often lack the ability to verify concurrent programs. In the past few years considerable advancement has been made in the verification of concurrent programs resulting in the development of verifiers such as Chalice and VeriFast [11, 15]. One of the most important developments is the use of separation logic to reason about concurrent programs [13]. Permission-based separation logic uses the concept that multiple threads can simultaneously have a read permission for a shared memory location whereas if one thread has a write permission for a shared location, no other thread can hold a read or write permission for that location. Permission-based separation logic is used in the VerCors toolset [1].

At the time of writing, the VerCors toolset is being developed. This toolset builds upon the Chalice toolset and it aims at verifying concurrent data structures, which are an important part of concurrent programs. Since verification of concurrent programs and data structures using permission-based separation logic is a relatively new area where substantial development is being made, there is demand for a case study in which a concurrent program is formally specified using permission-based separation logic. Using this formal specification, this program can later be verified using the VerCors toolset. Since the VerCors toolset is still under development and is currently not sufficiently developed to verify the concurrent program presented in this paper, this formal verification is not part of the research.

14th Twente Student Conference on IT January 24st, 2014, Enschede, The Netherlands.

The research question answered by this paper:

> *How can a model of a central printer server be formally specified?*

This question is answered in this paper using a case study of a printer server model. A printer server is a server to which users can send print jobs from one or more computers and from which users can retrieve and print their print jobs at one or more printers.

The main research question is divided into the following four sub questions:

1. What requirements exist for a central printer server?
2. How can the requirements for a printer server be expressed in a formal specification language?
3. How can a printer server be modeled?
4. Does the model respect the formal requirements for a printer server?

Related work on verification concepts, verification tools for sequential programs and verification tools for concurrent programs is presented in section 2. Section 3 gives the requirements for the case study in general, section 4 states the approach that was taken for this research and section 5 provides the requirements that were found for the system presented in this paper specifically. Section 6 presents the design of the developed program and in section 7 the fulfillment of the requirements is discussed. Finally, section 8 presents the conclusions of this paper and section 9 presents future work.

2. RELATED WORK

Substantial research has been done in the field of software verification. Most currently available verification tools are developed for verification of sequential programs. Verifiers such as Key, ESC/Java, Spec# and KIV can be used to prove non-trivial sequential programs correct [4, 5, 3, 12].

Over the past few years, focus in the development of program verifiers has shifted towards verifying concurrent programs. Verifiers such as Chalice, VeriFast and the VerCors toolset, which is currently being developed at the University of Twente, are aimed towards proving (parts of) concurrent programs.

Before going into the different concurrent verifiers, the concepts that these verifiers use are discussed.

Rely-guarantee reasoning is a well-known method for verifying concurrent programs. With rely-guarantee reasoning it can be specified that if certain properties hold at one point some other properties can be guaranteed to hold at a later point in the program. It has some important disadvantages however, such as the requirement that the rely and guarantee conditions for a module need to specify all shared resources even if some of those resources are not used by that module. Another disadvantage is the necessity for shared resources to be globally known, which makes having dynamically allocated shared resources rather difficult [9]. These disadvantages make rely-guarantee reasoning too complicated to integrate directly in existing tools for sequential verification [1]. Advancements have been made to combine rely-guarantee reasoning with separation logic in order to make it more local in the specification of rely and guarantee conditions, thus resolving the mentioned disadvantages [6].

Permission-based separation logic can also be used to ensure the correctness of concurrent programs. Permission-based separation logic is based on the concept that threads can have a permission in the domain (0,100] on a location at a given time. If a thread has a full permission for a location, i.e. the value of the permission is 100, then it has write-permission for that location. If a thread has a partial permission for a location, i.e. a value in the domain (0,100), then it has read permission for that location [1]. The sum of all the permissions for a location is always equal to 100, which guarantees that when one thread has a write permission for a shared location, no other thread can hold a read or write permission for that location. This also guarantees that multiple threads can simultaneously have a read permission for a shared memory location as long as their combined permissions do not exceed 100.

Recently, additions have been made to separation logic that make it possible to specify a read permission on a shared location without having to explicitly state a concrete number (between 0 and 100) as a value for that permission [7].

An especially important part of separation logic is locking. The permissions that a lock protects are specified with a special predicate named `inv` associated with that lock. The predicate is defined by the client that uses the lock and is sent to the lock class through a class parameter. The `inv` predicate, and therefore the associated permissions can then be acquired through acquiring the lock. These permissions are then released when the lock is released. An example of a specification that uses a lock is given in Listing 1. Here the lock holds a write (and read) permission for the `resource` variable.

In the example in Listing 1, the `inv` predicate is equal to the full permission for `resource`. This predicate is associated with the lock on line 6. On line 7, with the special `commit` command, the permissions are stored inside the lock and therefore no longer belong to the current thread.

Listing 1: Lock specification

```
1 private Lock[] lock;
2 private int resource;
3 ...
4 init() {
5   /*@ pred inv = Perm(resource, 100); @*/
6   lock = new Lock/*@<inv>@*/();
7   //@ lock.commit();
8 }
```

In Listing 2 the specification for a `Lock` class is shown. The `lock()` method requires no permissions and returns the permissions associated with the `inv` predicate [8]. The `unlock()` method requires the permissions associated with the `inv` predicate and returns no permissions at all. Through these two methods, other classes can acquire the permissions this lock protects and return them to the lock after using them.

Listing 2: Lock specification

```
class Lock {
  /* @requires true;
   * @ensures inv; */
  void lock() {}
  /* @requires inv;
   * @ensures true; */
  void unlock() {}
}
```

In practice, `ReentrantLock`s are often used in Java programs. The specification of such locks differs somewhat from the specification in Listing 2 because when the thread holding a lock, locks on that acquired lock once more, it should not get any additional permissions and similarly, only the final unlocking should return the protected permissions to the lock. The exact difference in specification between `ReentrantLock`s and simple `Lock`s is beyond the scope of this paper.

The Chalice verifier [11] uses a concept similar to permission-based separation logic and a limited form of rely-guarantee reasoning. Chalice can verify programs written in its own programming language using Boogie internally. Boogie [2] was originally developed together with the Spec# sequential verifier. Boogie is an intermediate verification language which can be used to abstract from different source languages [10].

VeriFast [15] can be used to verify concurrent Java programs. The method contracts used to annotate the Java source files should be written in a form of separation logic combined with inductive data type and fix-point definitions, lemma functions and proof steps in order to use VeriFast to verify the program.

The VerCors toolset [1] uses Chalice and Boogie internally. VerCors first converts the Java program and its annotations into Chalice's verification language and then uses Chalice's translation to Boogie to generate the proof obligations. As annotation language JML is used, extended with syntax to express permission-based separation logic contracts. This keeps the specified contract concise and readable and since JML is a widely used specification language, the time and effort required for learning how the specification should be written is minimized.

The aim for the VerCors toolset is to make verification of a larger set of shared data structures possible compared to other concurrent verifiers. The VerCors toolset also aims at a higher level of automation in writing the specification than VeriFast because VeriFast requires a user to write many intermediate assertions explicitly [1].

The discussed related work consists mostly of concurrent verification tools and the concepts they adopt. This research adds upon this by developing (part of) a reasonably

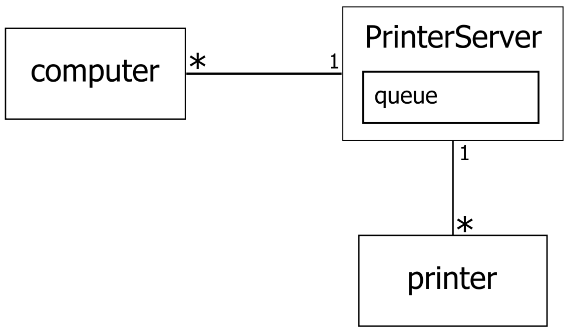

Figure 1: High-level design

complicated concurrent program that is formally specified, which can be used by future concurrent verifiers to show the working of that verifier.

3. CASE STUDY REQUIREMENTS

Some requirements had to be taken into account when deciding on the kind of program and on the scale of the program that has been designed, specified and implemented for this research. This section will discuss those requirements, whereas the requirements for the specific program itself are presented in section 5.

Since the program for this case study is to be used to test concurrent verifiers in the future, the program ought to be multi-threaded. During the development of these verifiers, small test programs are written by the researchers. For this case study to have added value, it needs to use a somewhat larger or more extensive program. Lastly, the program for this case study should be as compatible as possible with concurrent verifiers currently being developed.

A reasonably complex multi-threaded Java program specified with an extension of JML that is supported by the VerCors toolset, fulfills these requirements. A central printer server has therefore been designed, specified (in extended JML) and implemented (in Java). In the next section the approach taken to obtain this program is described.

4. APPROACH

Figure 1 shows a very high level design of a central printer server. The picture shows that zero or more computers can send print jobs to a printer server from which zero or more printers can then retrieve those print jobs to print them. The printer server has one queue that holds the print jobs after they are added and before they are retrieved. This is where data races could happen and this is the focus of this case study.

Java classes have been designed and implemented to model a central printer server. These classes have been supplemented with a formal contract expressed in the Java Modeling Language (JML) compatible with the VerCors toolset. The formal specification of the printer server model has consequently been used to discuss how this contract ensures the fulfillment of the general requirements of a printer server.

The first step of the research is to formulate the requirements for the printer server. These are then used in the design of the printer server program. Especially the data structure that is used in the program has been selected with the program's requirements in mind since this is the part of the program where most possibilities for interference occur.

Simplicity has been an important factor in the design of the printer server. The main idea has been to specify a larger and more complex program with a shared data structure that had not been specified before. The aim was to refrain from designing and implementing an unnecessarily complex program as this would complicate the formal specification and the proving of (parts of) the program, while simultaneously maintaining the possibility to generalize the simple classes and concepts to more complex ones.

The program's implementation is based on the program's specification which was formulated after the design had been completed. The formal specification was amended and supplemented when new insights occurred during the implementation.

The presented approach answers sub-research questions 2 and 3, which respectively question how the requirements for a printer server can be expressed formally and how a printer server can be modeled.

The developed concurrent program with its formal specification is expected to be usable to test concurrent verification tools such as the VerCors tool when the development of such tools has progressed sufficiently.

5. PRINTER SERVER REQUIREMENTS

The functional requirements for the printer server are presented below. The requirements are given in natural language.

Sub-research question 1 asks what requirements exist for a printer server. The requirements stated below answer this question. The requirements are valid for any central printer server and they are some of the most fundamental requirements.

1. A user can only see his own print jobs.
2. A user can only print his own print jobs.
3. Print jobs cannot get lost.
4. It should not take too long for the user to be able to see an added print job.
5. A user should be able to select which print job he wants to be printed.
6. A print job cannot be printed more than once.
7. Once a print job is printed it should not be viewable anymore.
8. The printer program should be data race-free.
9. The printer program should be deadlock-free.

These requirements influence design choices for the printer server program.

6. PROGRAM DESIGN

A central printer server normally receives print jobs from different computers, possibly from different users. Users can retrieve their own print jobs at a number of printers that are connected with the printer server. The possibilities for interference depend on the choice of data structure for storing the print jobs at the server.

Since different users should not be able to access each other's print jobs, the possibilities for interference can be minimized by giving each user a distinct part of the printer

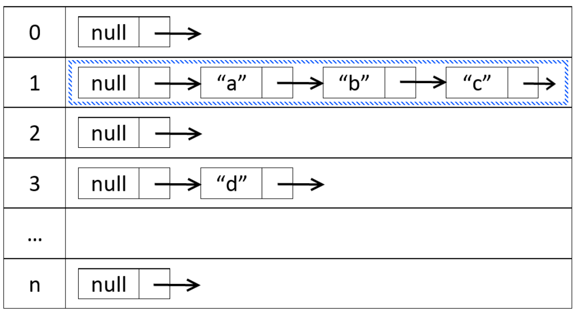

Figure 2: The printer server's queue: an array of linked lists

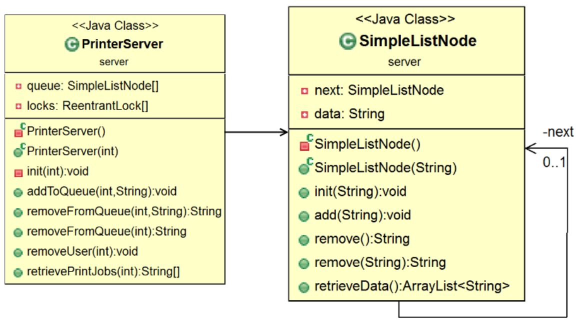

Figure 3: Class diagram of the central printer server

server's queue. Different users are then enabled to access the queue simultaneously without further synchronization. Interference might still occur, typically when at approximately the same time: a user sends print jobs from multiple computers, a user prints print jobs at multiple printers or a user sends a print job and prints a print job. These kinds of interference can then only occur in the part of the queue belonging to one user.

From this analysis, it becomes clear that a lock should protect at most one user's part of the queue because the user only needs access to his own print jobs and since no other user needs access to his print jobs.

Locking every print job separately is possible and this would enable the user to do the actions described above, such as sending a print job to the queue and simultaneously printing one. This would however complicate the specification and implementation of the printer server because more locks are needed in a less trivial way compared to locking the whole user's part of the queue. The cases that would be enabled by more fine-grained locking can be considered marginal. For example, a user would normally print his print jobs at one printer at a time, not two and a user that would simultaneously add a print job from his computer and print one at a printer would most likely want to print the most recently added print job so he would not mind waiting for that print job to be available.

Therefore, a locking strategy was chosen where locks protect at least and at most the part of the queue that belongs to a specific user, a user queue.

A data structure similar to a hash table whereby locking happens per row of the table, was found suitable. Each row would then store the print jobs belonging to a specific user. For simplicity, an array of linked lists is used as the printer server's queue as depicted in Figure 2. Each slot in the array has a lock associated with it which holds all permissions for the linked list at that position of the array. In the figure, everything within the blue dashed rectangle is protected by one lock.

A class diagram of the two classes designed for this case study is given in Figure 3. The `SimpleListNode` class has been implemented as a bare-bones class containing the basic functionality necessary to represent a linked list.

Because each lock protects a complete row of the queue, locks are specified and used in the `PrinterServer` class. Methods in `SimpleListNode` that need permissions for one or more attributes protected by a lock, require having permissions to the whole linked list before the method is entered. They return these permissions to the calling class after the method returns, in turn enabling `PrinterServer` to return the permissions to the lock.

Every slot in the array that represents the printer server's queue is always associated with an initialized `SimpleListNode`. The first `SimpleListNode` of each user queue has `null` as data and every following `SimpleListNode` contains a print job. In the simple program presented here, print jobs are represented through simple `String`s, although this could easily be extended to more complex classes.

In the printer server's queue, it is assumed that each user has a unique identification number (id). This number is used as an index in the array representing the printer server's queue to find the linked list containing that user's print jobs. In a more realistic model, a hash of the user's id might be used as the index rather than the id itself. Because hashing the id first does not significantly change any of the concepts of the program, it was deemed unnecessary to include this in this case study.

Since all user queues are initialized beforehand, the maximum number of users is also set beforehand in the presented program. Adding and deleting user queues to allow more flexibility in the number of users, would be a significant change for the program's design. However, this would also add considerably more complexity without making the program much more realistic since a predetermined maximum number of users could suffice for many printer servers as well.

In the array of locks in the `PrinterServer` class, the lock at a certain index in that array protects the linked list at that same index in the printer server's queue. For example, the lock at index 1 in the array of locks protects all fields enclosed by the blue dashed rectangle in Figure 2.

7. REQUIREMENTS FULFILLMENT

In section 5 some requirements for a central printer server have been presented. In this section the fulfillment of these requirements will be discussed using the specification that was written for the program. The fulfillment of some of the stated requirements can be deduced from informal functional specifications, whereas other requirements can be shown to be fulfilled using formal specifications about permissions.

7.1 Fulfillment of requirements through informal specification

Requirement 1 and 2 state that a user should only be able to see and print his own print jobs. These requirements are currently not fulfilled by the program since there is no required authentication when retrieving a print job. These requirements could be fulfilled by implementing an authentication mechanism that uses a user's id and a personal password to authenticate a user before granting access to a user queue.

Requirement 3, which states that print jobs cannot get lost, has to do with requirement 8 on data races because print jobs could get lost through data races. However, in the next subsection it is explained that the absence of data races can be guaranteed. Furthermore, the method `add` in `SimpleLinkedList`, which is stated in Listing 4, is responsible for adding print jobs to the queue. Lines 7 through 10 of Listing 4 state that when this method ends, the last element in the list indeed contains the added print job, that the length of the user queue is increased by one and that there now exists one more node containing `elementData` as data, as compared to before the method was entered.

The ensures given on lines 7 through 10 make use of the model methods `containsLast`, `size` and `count`. These model methods are specified in JML and they can only be used in the specification, not in the Java code. The model method `containsLast(String)` returns a boolean depicting whether the last node contains the given string, `size` returns the size of this linked list and `count(String)` returns the number of occurrences of the given string.

Listing 3 shows that adding a print job starts with acquiring a lock on a user queue. Listing 4 shows that after that the end of the linked list is searched for recursively and a new node is appended to the end with the new print job. Lastly, the lock is freed as seen in Listing 3. As long as the user does not send a lot of add, remove or retrieve commands to his personal queue simultaneously, acquiring the lock should not take a lot of time since no other user has a reason to acquire this lock. The time it takes to add the print job to the queue depends on the length of this user queue. If it is not incredibly long, this simple operation should be very quick. Lastly, unlocking takes very little time. Therefore, in almost all cases a newly added print job will be visible very soon after adding it which means requirement 4 is satisfied.

The `PrinterServer.retrievePrintJobs` method allows a user to retrieve all print jobs currently in a user queue. The `PrinterServer.removeFromQueue` method can then be used to remove a specific print job by giving the user's id and the print job that should be removed as parameters. Together these methods can be used to select a specific print job to print, thereby fulfilling requirement 5.

Requirement 6, a print job cannot be printed more than once, and requirement 7, once a print job is printed it should no longer be viewable, can be guaranteed as long as the `PrinterServer.retrievePrintJobs` method is only used to present the user the choice which print job he wants to print. Then the `PrinterServer.removeFromQueue` method can be used to print the print job whilst also removing it from the queue, thus ensuring it cannot be printed or viewed anymore.

Because the program uses locks, deadlock-freeness is a requirement that should be considered. A thread does not need to acquire more than one lock at a time, since it does not need to modify multiple user queues simultaneously. Therefore, there is no possibility for deadlocks and requirement 9 is fulfilled.

7.2 Fulfillment of requirements through formal specification

The fulfillment of requirement 8 on data race-freeness is ensured by formal specifications for all shared data structures in the program. As an example, the method `addToQueue` in the `PrinterServer` class is given along with its specification in Listing 3. Below, this method will be analyzed in detail to show how a specification can be used to give guarantees on the working of a program.

The full code of the implemented classes with their specification can be found on the website of the Formal Methods and Tools department of the University of Twente [14].

Listing 3: `PrinterServer.addToQueue`

```
1  /**
2   * Adds a PrintJob to the user's queue.
3   * @param printJob The added print job.
4   * @param id The ID of the user.
5   * @requires printJob!=null ** id>=0 **
6   *   id<queue.length ** queue[id]!=null **
7   *   Perm(id,p) ** Perm(locks[id],p) **
8   *   Perm(printJob,p);
9   * @ensures queue[id]!=null **
10  *   Perm(locks[id],p);
11  */
12 public void addToQueue(int id, String
      printJob) {
13   locks[id].lock();
14   queue[id].add(printJob);
15   locks[id].unlock();
16 }
```

Next, some specifications relevant for proving the `addToQueue` method are given, followed by the proof outline of that method.

7.2.1 *Specification of* `addToQueue`

The `addToQueue` method requires the given print job to actually exist and it requires `id` to be a valid index in the `queue` array. The latter requirement exists because in this simple model, a user's id is the place in the array where that user's personal print queue is stored. The method also requires `queue[id]` to exist and it requires read permission to `id`, `locks[id]` and `printJob`. These read permissions are expressed through writing `Perm(attribute, p)`, meaning that the thread requires permission p for `attribute`. Here p stands for partial, meaning any permission between 0 and 100. The verification tool can decide the exact number for the permission.

The `addToQueue` method needs read permission to `id` and `printJob` in order to use these parameters in the method body. It also needs read permission to `locks[id]` to be able to call `lock()` on it.

Before going into locking, the call to the `add` method is discussed. The variable `queue` is an array of `SimpleLinkedLists` and `queue[id]` therefore points to one such a `SimpleLinkedList`. The method `SimpleLinkedList.add` which is called on line 14 of Listing 3, is provided in Listing 4.

Listing 4: `SimpleLinkedList.add`

```
1  /**
2   * Adds given data to end of the list.
3   * @param elementData The added data.
4   * @requires Perm(elementData,p) **
5   *   elementData!=null ** nodeState();
6   * @ensures nodeState() **
7   *   containsLast(elementData) **
8   *   size() == \old(size())+1 **
9   *   count(elementData) ==
10  *   (\old(count(elementData))+1);
11  */
12 public void add(String elementData) {
13   if(next == null) {
14     next = new SimpleListNode(elementData);
15   } else {
    next.add(elementData);
  }
}
```

The method **add** requires the data for the element that will be added to exist, it requires read permission to it and it requires to have all the permissions included in the `nodeState` predicate, which is defined in the class SimpleListNode. A predicate can be seen as the JML equivalent of a function and the `nodeState` predicate is provided in Listing 5. The predicate includes read and write permission for the `data` attribute, for the `next` attribute and if a next node exists, it requires the same permissions for that next node resulting in a recursive call to `nodeState`. Thus, `nodeState` includes read and write permission to the `data` and `next` attributes of this node and all nodes following this node until the end of this linked list.

Listing 5: The `SimpleListNode.nodeState` predicate

```
/*@
 * public resource nodeState() =
 *   Perm(data,100) ** Perm(next,100) **
 *   ((next!=null) -* next.nodeState());
@*/
```

The **add** method in Listing 4 requires more permissions than the `addToQueue` method in Listing 3. Therefore, the additional permissions have to be obtained inside the `addToQueue` method before it can call `queue[id].add`. These additional permissions are obtained through locking. On line 13 of Listing 3, `locks[id].lock()` is called. Through this call, the permissions protected by that lock are acquired. In section 2 about Related Work, in the part about permission-based separation logic, the concept of locking is explained.

Listing 6 provides the `state` predicate which is the predicate that is associated with each of the locks in the array of locks in the `PrinterServer` class. The lock at index i of this array is associated with `state(i)`. Thus the lock at index i protects the permission to read and write `queue[i]` and if `queue[i]` exists, it protects the permissions associated with the `nodeState` predicate in `queue[i]` too.

Listing 6: The `PrinterServer.state` predicate

```
/*@
 * public resource state(int i) =
 *  Perm(queue[i],100) **
 *  ((queue[i]!=null)-*queue[i].nodeState());
@*/
...
for(int i = 0; i < queue.length; i++) {
  queue[i] = new SimpleListNode(null);
  locks[i] = new ReentrantLock/*@<state(i)>@
    */();
  //@ locks[i].commit();
}
```

7.2.2 *Proof outline*

To prove the `addToQueue` method correct, the requirements for the **add** method can be assumed given and the method can then be analyzed line by line to show that after the method the ensured statements hold. The outline for such a proof is provided in Listing 7.

Since the requirement that `id` is a valid index in the `queue` array and the requirement that `queue[id]` exists are less interesting, they are left out of the proof outline for improved readability. Furthermore, coloring is used for more clarity. The lines with executable Java code are in black, the lines with commands clarifying the actions that are performed on the permission statements are in green, statements that are newly added are in red (this happens through locking, after the return of a called method or after folding/unfolding predicates) and statements in bold are relevant for the next line. All other statements are in gray.

In Listing 7 the actions performed on the permission statements (stated in green) each have a different meaning. 'Reordering' means that the statements are ordered differently. This is stated explicitly to avoid confusion. 'Unfold' means that a predicate is being unfolded, thus replacing the left side of a predicate definition with the right side. 'Fold' has the opposite meaning and during folding the right side of a predicate definition is replaced by its left side. Lastly, 'Weakening' depicts the removal of one or more statements, thereby lessening the permissions stated in the proof outline. Weakening is applied in the given proof outline when a permission is irrelevant for obtaining the intended result (the ensured permission statements after the method).

Listing 7: Proof outline for `PrinterServer.addToQueue` method.

```
 1  {printJob!=null ** Perm(id,p) ** Perm(locks[
        id],p) ** Perm(printJob,p)}
 2  locks[id].lock();
 3  {printJob!=null ** Perm(id,p) ** Perm(locks[
        id],p) ** Perm(printJob,p) ** state(id)}
 4  ( Unfold )
 5  {Perm(printJob,p) ** printJob!=null ** Perm(
        id,p) ** Perm(locks[id],p) ** Perm(queue[
        id],100) ** ((queue[id]!=null) -* queue[
        id].nodeState())}
 6  ( queue[id]!=null is true )
 7  {Perm(printJob,p) ** printJob!=null ** Perm(
        id,p) ** Perm(locks[id],p) ** Perm(queue[
        id],100) ** queue[id].nodeState()}
 8  ( Reordering )
 9  {Perm(printJob,p) ** printJob!=null ** queue[
        id].nodeState() ** Perm(queue[id],100) **
          Perm(id,p) ** Perm(locks[id],p)}
10  queue[id].add(printJob);
11  {queue[id].nodeState() ** Perm(queue[id],100)
          ** Perm(id,p) ** Perm(locks[id],p)}
12  ( Fold )
13  {state() ** Perm(id,p) ** Perm(locks[id],p)}
14  locks[id].unlock();
15  {Perm(id,p) ** Perm(locks[id],p)}
16  ( Weakening} )
17  {Perm(locks[id],p)}
```

The statements printed in bold on line 9 of Listing 7 are the requirements for the `SimpleListNode.add` method. The statement in red on line 11 are the statements ensured by this method. Assuming that **add** complies with its contract, these permissions hold after the method returns.

Listing 7 clearly starts with the preconditions of the `addToQueue` method and it ends with the postconditions of that method. Therefore, `addToQueue` is correct.

An automated verifier could prove every method this way, whilst also checking that methods only ever read from or write to attributes for which they have the applicable permissions. The verifier can then guarantee that the program is data race free.

8. CONCLUSION

This paper has shown which requirements exist for a central printer server, it has discussed that these can be expressed in a formal specification language such as an extended version of the Java Modeling Language and it has explained how a Java program modeling a central printer server has been designed, specified and implemented. Finally, this paper has also presented the outline of a proof showing that the program complies with the specification given for it.

The aim of this research has been to develop a program that can be verified by concurrent verifiers in the future, specifically by the VerCors toolset.

The full code of the implemented classes with their specification can be found on the website of the Formal Methods and Tools department of the University of Twente [14].

9. FUTURE WORK

In this case study, a significantly larger concurrent program to test concurrent verifiers with, has been presented. However, future research in this area could discover more specific requirements for programs that can be used for testing concurrent verifiers.

Research could be done on the state of different existing concurrent verifiers to determine their strengths and weaknesses. The results thereof could then be used to improve and extend these verifiers or to develop new verifiers.

10. REFERENCES

[1] A. Amighi, S. Blom, M. Huisman, and M. Zaharieva-Stojanovski. The VerCors Project: Setting Up Basecamp. In *Sixth Workshop Programming Languages meets Program Verification, PLPV 2012*, pages 71–82, New York, Jan. 2012. ACM.

[2] M. Barnett, B.-Y. E. Chang, R. DeLine, B. Jacobs, and K. R. M. Leino. Boogie: A modular reusable verifier for object-oriented programs. In *Formal methods for Components and Objects*, pages 364–387. Springer, 2006.

[3] M. Barnett, K. R. M. Leino, and W. Schulte. The Spec# programming system: An overview. In *Construction and analysis of safe, secure, and interoperable smart devices*, pages 49–69. Springer, 2005.

[4] B. Beckert, R. Hähnle, and P. H. Schmitt. *Verification of object-oriented software: The KeY approach.* Springer-Verlag, 2007.

[5] D. R. Cok and J. R. Kiniry. Esc/Java2: Uniting Esc/Java and JML. In *Construction and Analysis of Safe, Secure, and Interoperable Smart Devices*, pages 108–128. Springer, 2005.

[6] X. Feng. Local rely-guarantee reasoning. In *Conference Record of the Annual ACM Symposium on Principles of Programming Languages*, number October in 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL'09, pages 315–327, Toyota Technological Institute, Chicago, IL 60637, United States, 2009.

[7] S. Heule, K. R. M. Leino, P. Müller, and A. J. Summers. Fractional permissions without the fractions. *Proceedings of the 13th Workshop on Formal Techniues for Java-Like Programs - FTfJP '11*, pages 1–6, 2011.

[8] D. Hutchison and J. C. Mitchell. Programming Languages and Systems. pages 171–186, Bangalore, India, 2008. Springer.

[9] C. B. Jones. Tentative Steps Toward a Development Method for Interfering Programs. (October):596–619, 1983.

[10] K. R. M. Leino. This is Boogie 2. *Manuscript KRML*, 178(June), 2008.

[11] K. R. M. Leino, P. Müller, and J. Smans. Verification of Concurrent Programs with Chalice. In *Proceedings of the 11th international conference on Verification, Model Checking, and Abstract Interpretation*, VMCAI'10, Berlin, Heidelberg, 2010. Springer-Verlag.

[12] W. Reif, G. Schellhorn, K. Stenzel, and M. Balser. Structured specifications and interactive proofs with KIV. In *Automated deduction — a basis for applications*, pages 13–39. Springer, 1998.

[13] J. Reynolds. Separation logic: A logic for shared mutable data structures. In *Logic in Computer Science, 2002. Proceedings. 17th Annual IEEE Symposium*, volume 1, pages 55–74. IEEE, IEEE Comput. Soc, 2002.

[14] M. Roo. Full code and specification for a printer server case study. `http://fmt.ewi.utwente.nl/education/bachelor/144`, January 2014.

[15] J. Smans, B. Jacobs, and F. Piessens. Verifying Java programs with VeriFast. *Aliasing in Object-oriented Programming*, (Section 2):1–18, 2012.