Specifying a Concurrent Program in Java using Separation Logic

Dennis van der Zwet
University of Twente
P.O. Box 217, 7500AE Enschede
The Netherlands
d.vanderzwet@student.utwente.nl

ABSTRACT

When building software to be used in critical environments, it is important that the software behaves as intended. Permission based separation logic is a way to specify a concurrent program so that the behavior can be verified. In this paper we describe the building of a ski lift program and specify it using permission based separation logic. We use this case study to see what problems arise when trying to use separation logic to specify a realistic program.

This paper starts with an introduction into permission based separation logic, after which it describes the building of the ski-lift program. After this it explains how we specified the program. We explain which choices we made and why we made them, after which we explain which problems we found while specifying the program and what solutions we used to solve the problems.

Keywords

Prototype, proof of concept, VerCors, specifying concurrent programs.

1. INTRODUCTION

When a computer is used to perform tasks, it is the intention that the the computer behaves in a certain way. When a computer is used to perform safety-critical applications, it is important that the software functions *exactly* as intended. For example when a radiographic image is taken, it is important that the correct dose of radiation is used, since too high doses can cause human casualties.

Because of this, when writing software, it is important that a program has correct behavior. To define what the correct behavior for a program is it is possible to create a specification of the software. While it, is possible to specify a program using an informal specification, such as human language, there are several problems. Since human language can be ambiguous and hard for a computer to understand, it is hard to say when a program does not follow its specification.

This is the reason that in the process of verifying the program behavior, the program is usually specified using formal specifications. In a formal specification the behavior of a program is specified using specific formulas, for example using mathematical logic.

For sequential programs, there are techniques to define a formal specification of how the program is supposed to work, such as Hoare logic, which is further explained in Section 2. There are also tools and methods to verify that a program conforms to the given specification. When verifying a formal specification, it is possible to let a computer check if the program conforms to the specifications. Usually this involves defining pre- and postcoditions on a method. A precondition is a set of conditions the state of a program has to adhere to before the execution of the method, and a postcondition gives guarantees about the state of the program after the method. In Section 2 specifying and verifying programs is explained more. While those ways of verifying programs are intended for sequential programs, many programs now have some form of concurrency.

The reason for the increased concurrency in programs is that in the past few years the improvements in hardware performance that is usable for sequential programs are getting slower. A large part of the performance increases are gained by adding multiple cores to a single processor, and adding more processors to certain systems. To make use of these extra resources, programs have to make use of those extra cores. These so called concurrent programs use multiple threads to split the workload over multiple cores

With concurrent programs, specifying and verifying the behavior is more difficult. Since in concurrent programs the order in which concurrent parts of the program are run is not predictable, there are extra hurdles in verifying that a program does not violate a specification. For example when incrementing a counter by one, in a non-concurrent program, the precondition could be that the counter must be initialized, and the postcondition could be that the counter will be 1 higher than before the method was called. In a concurrent example this would not be sufficient, since in a concurrent program it is possible that two threads increase the counter at the same time, which could mean that by the end of one of the increase methods, the counter is 2 higher compared to the beginning, which will break the postcondition, since that ensured it would be exactly 1 higher.

In this research we are going to use permission based separation logic to specify multithreaded programs, which will be explained in Section 2. We are going to develop a program which models a ski-lift using multiple types of cars, which we will explain in Section 3. We chose this program because when a ski-lift malfunctions, it could have serious implications, therefore it is important for a ski-lift to be working correctly. After developing the model of a ski-lift, we will specify it using permission based separation logic to see

22nd Twente Student Conference on IT, January 23, 2015, Enschede, The Netherlands.

what problems arise when trying to use separation logic to specify a realistic program.

The research question that will be addressed will be
"How difficult is it to specify a multithreaded model of a complex ski-lift using permission based separation logic?"

This question will be broken down into the following subquestions:
"How can the proposed model be built in a concurrent way such that it serves as a good use case to be specified using permission based separation logic?"

"What problems arise when trying to specify the built ski-lift program using permission based separation logic so it can be used to eventually verify the program?"

"How can the problems that arise be solved?"

2. BACKGROUND

2.1 Methods of verification

A lot of work has been done to verify the correct behavior of programs. The verification that is used nowadays builds on Hoare[4] logic. Hoare logic uses the concept of preconditions and postconditions to reason about code. Hoare defined a set of rules used to prove if a method returns in a state that conforms to its specification. To prove this it specified preconditions and postconditions. A precondition is a set of conditions, which must be adhered to before using a method. A postcondition is a guarantee that if at the beginning of a method call the preconditions are met, the state of the program after the usage of the method conforms to the conditions set in the postcondition. If the postcondition can be proved using the precondition and the actual code, it is verified that the method behaves as specified.

Because in a sequential system there is no other process modifying the variables used by the program, it is possible to prove that the outcome will adhere to the specifications.. However, since in concurrent programs it is possible that data is accessed and modified by another thread during the execution of a method, the set of rules in Hoare logic is not sufficient to prove the adherence to the postconditions.

One of the first methods to verify behavior of concurrent systems is researched by Owicki and Gries [6]. They used a method called rely-guarantee. While it is possible to specify a program using rely-guarantee, it provides a considerable specification overhead. Since this makes it impractical to use on a large scale, alternatives have been researched. An alternative that has been researched is separation logic.

2.2 Separation logic

Separation logic is an extension of Hoare logic which distinguishes between the use store and a heap in the logic[1]. When separate threads do not communicate with each other, the heaps of the threads will be disjoint, which automatically means no data races will occur, since they will not try to use the same variables. Since usually it is necessary for different threads to communicate during execution of those threads, this is too restrictive. Permission based separation logic adds permissions to the logic.

In permission based separation logic, every use of memory goes with a certain permission to use that memory, which is a number in the range (0,1]. A write operation needs a permission level of 1, while a read operation can have a permission of every number in this range. The sum of all permissions on a variable cannot be higher than 1. This enforces that a write operation can only take place in the case that it has a permission of 1, while the constraint that the total permission of 1 makes sure no other piece of the program can read that piece of memory.

Nowadays there are multiple techniques to verify concurrent programs that use verification logic. Many ways to verify concurrent programs are either based upon the work of Owicki and Gries or using separation logic, for example VerCors[2] and Verifast[5]. We will go in detail about the technique we used, namely VerCors.

2.3 VerCors

A tool that is in development is called VerCors. It uses *permission based* separation logic for the specification of a program, and verifies if it follows the specification. It uses the constraints specified in the specification and checks if the following code actually adheres to those constraints. For this it uses the Java modeling language (JML) which it extends with additional functionality.

JML is a language used to add the specification of a Java program in the comments of a program. JML is specified in comments by using an '@' sign followed by a keyword. Keywords are for example *requires* and *ensures*, which respectively define the precondition and postcondition of the following code fragment.

Vercors extension to JML adds the possibility to reason about permissions and separation in the JML. This is done by adding the Pointsto(x,v,π) and Perm(x, π) predicates. Where x is the memory to point to, v the value of this memory and π the permission level. Of these methods we will use the Perm method, which means a permission on x of the value π. So when it requires a Perm(x,50), it requires a read permission on x, and when it ensures a Perm(y,100), it gives back a certan write permission.

2.3.1 Example

Below follows a very simple example showing a counter which has a method to increase the value of the counter. Because the method writes to value x, it needs write permission, which is why it needs a permission of 1.

Note that in the JML instead of a value from 0 to 1 it is an integer between 0 and 100. As shown in this example, the constructor only has a postcondition, in which it ensures a permission. When the constructor creates the object, it has control over the variables in the object. The postcondition specifies that it indeed has the control over the variable x, since it is created in the constructor.

The method increase requires a permission of 100, which means a write permission on x, since it has to increment it. After the usage of the increase method it ensures the permission of 100 again, which means it passes the permission back. In this method \old is defined in the standard JML as the state of the program before the method started, so old(x) is the value of x before it was increased by the method.

```
class Counter{
private int x;

//@ensures Perm(x,100) && (x==0);
public Counter(){
    x = 0;
```

```
}

//@requires Perm(x,100);
//@ensures Perm(x,100)
&& (x == (\old(x))+1)  ;
public void increase(){
    x = x+1;
}}
```

Listing 1: An example of separation logic

3. SKILIFT PROGRAM

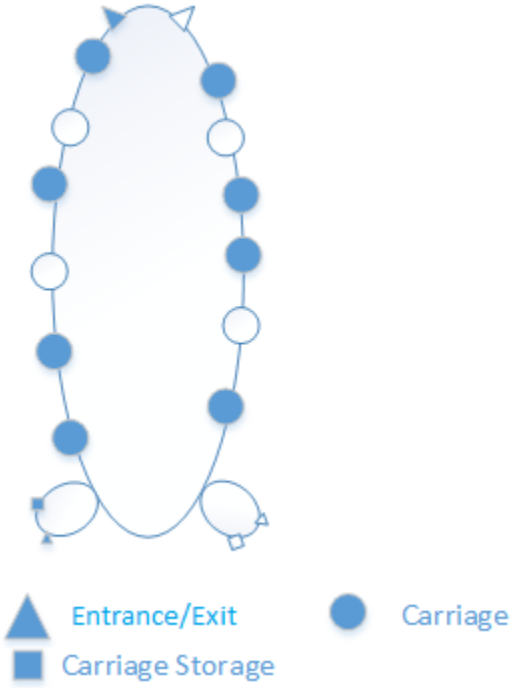

Figure 1: schematic example of ski-lift

The program to be specified is a model of a hybrid ski lift, which is a ski lift which has multiple types of cars, for example a gondola for the beginners, and chairs for more advanced skiers. The reason to go for a ski lift is that it is a safety critical scenario. Since a ski lift is a form of human transportation, safety is very important.

There are people who want to get on one of the types of cars to the top of the ski lift, after which they get off. To get in a car they join the queue for that specific type of car. When a car is not needed, it sits in storage, and when it comes out of storage, it goes to the queue, takes passengers, after which it goes on the main cable to the top. There is a minimum amount of space needed between cars, and the cables also have a maximum amount of cars allowed on it.

The main cable will not stop moving as long as the program is running, while the cables that are attached to the queues will wait when a car is at the point to get attached to the MainCable, but the MainCable does not have a space.

The program is built up in different classes, which is show in the class diagram in Figure 2. In the next section we will explain the functionality of the classes.

Figure 1 shows a schematic example of a ski lift. In this example the round shapes are the cars, the triangles represent the entrances and exits, and the squares represent the storage of the cars. The different colours of the shapes, filled blue and blue with white filling represent two different types of cars.

The code of the program can be found on the site of the University of Twente[9]

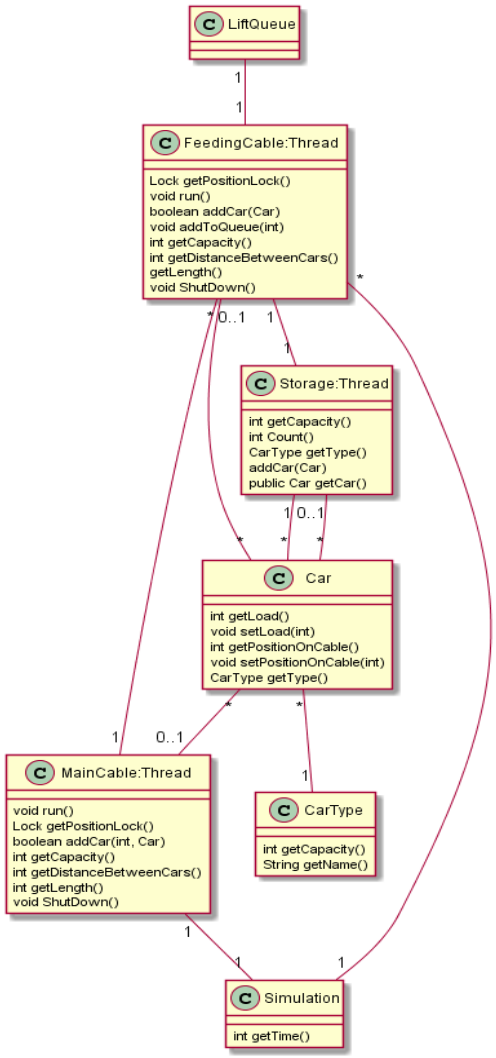

Figure 2: Class diagram of the ski lift program

3.1 Car

The car datatype only holds information about itself, and does not know anything about the rest of the world, except its position on the cable it is on. Which cable is not important. It has setters and getters to get and set the position on the cable, and how many people are in the car. It also has a CarType

3.2 CarType

A CarType has a name, to differentiate between the different types of cartypes, and a capacity, which determines how many people can be in a car of the specific type. The name of a CarType needs to be unique.

The reason that this construction is used instead of a car interface which gets implemented by specific types of cars is that since there is no logic specific to the carTypes, and the only difference is the name and the capacity, there would be no clear advantage to use this approach, while there would be the disadvantage of needing more classes for every time an extra type of car is needed.

3.3 Simulation

The simulation class only provides a timer, which starts at 0 at the start of the program, and makes sure all the cables are on the same time.

3.4 MainCable and FeedingCable

The *MainCable* and *FeedingCables* are somewhat similar. Both have cars that ride the cable, which needs to change position. It would have been possible to make a superclass *Cable* which were inherited by *MainCable* and *FeedingCable*, but while some parts work the same, all the functionality has a different implementation, and also while using the cables, they should not be used on the same way.

The most important part of both of the cables is the main loop, which uses the *Simulation* class to check if the *simulationTime* has been increased, and by how much, after which it moves the cars.

3.4.1 FeedingCable

The *FeedingCable* has 2 locks. The *positionLock* is used when the position of cars is altered, while the *queueLock* is needed to add or remove people from the queue.

The *moveCars* method of the *FeedingCable* first needs to check if the cable is allowed to move, after which it actually moves. It first loops through all cars that are on the cable, and if it should go to the *MainCable*, it checks if there is actually space on the *MainCable*, and adds a car if there is space by using the *addCar* method. If there is no space, the cable is not allowed to move. If the cable is allowed to move, it locks the *positionLock*, after which it moves every car one position, and releases the lock.

3.4.2 MainCable

The moveCars method of the MainCable does not have to check if it is allowed to move, since it will always move, so after it checks every car if it can be passed over the other cable, it moves all cars one position.

The *addCar* method of the *MainCable* checks if a car can be placed on a specific position. The difference between the *addCar* method of the *FeedingCable* is that it also keeps track of failed attempts to add a carriage. When a carriage is attempted to add, the position plus the time of the attempt is stored. When there actually is space on the cable, but another *FeedingCable* is waiting for a longer time, it will not be allowed to put a car on the cable, to avoid starvation of some of the *FeedingCables*.

A typical run of the application will be explained using a sequence diagram [Figure 3].

First the application needs to be initialized. In the sequence diagram this is somewhat simplified, but the administrator creates the application by creating a *MainCable* and the amount of *FeedingCables* it wishes to use, after which it adds the FeedingCables to the MainCable. After that it starts the threads of the FeedingCables and the MainCable.

When the application is running, if a user wishes to use the ski lift, he enters the queue, represented in the program by the AddToQueue method. After getting in the queue, the user gets in the carriage. In the program, the corresponding FeedingCables take care of putting users in the Cars, after which they get to the top, after which the user exits the lift. As shown in the sequence diagram, the order of which the different users get out of the lifts, does not necessarily have to be the same as the order in which they got in.

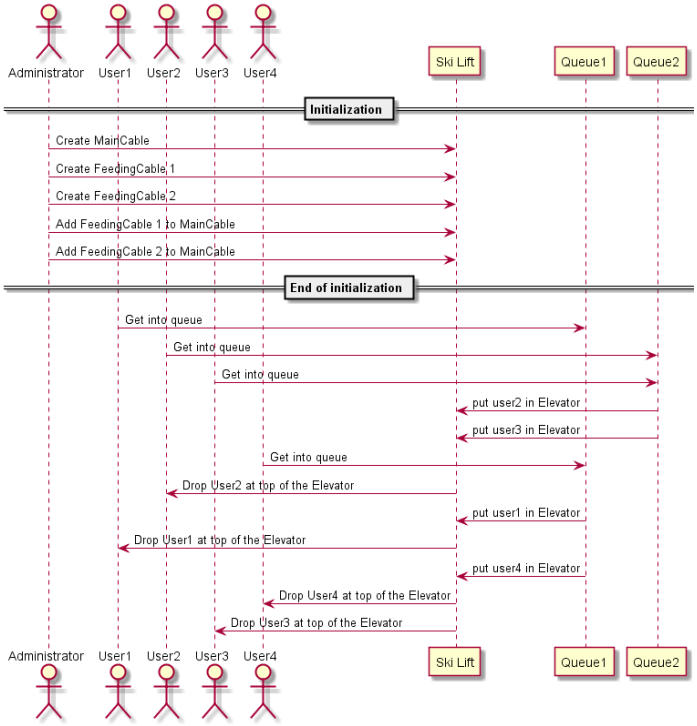

Figure 3: A sequence diagram specifying a run of the program with 4 people that wish to use the ski lift.

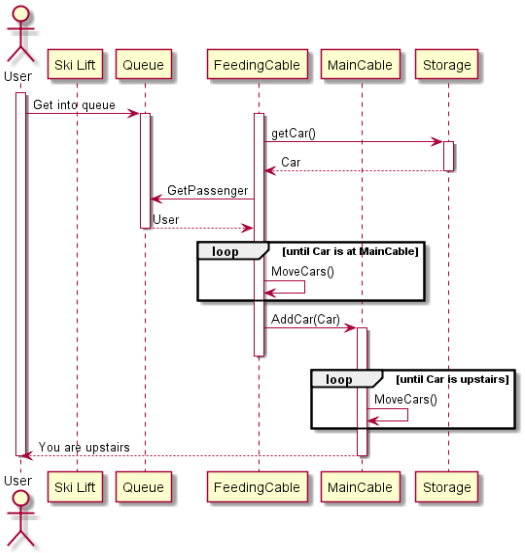

Figure 4: A sequence diagram specifying the trip of one person from the queue to the exit at the top of the ski-lift

A more detailed view of the program will be explained using the sequence diagram in Figure 4.

When a user enters the queue, the user will have to wait until he is at the top of the lift. The FeedingCable will check the queue for passengers every time a Car is at the position of the queue. It will then put passengers in the Car. After this the FeedingCable will call moveCars until the car is at the position of the MainCable. When the car is at the MainCable, it will add the car with the person in it to the MainCable. The MainCable also keeps calling moveCars, until the car is at the top, at which point the user exits the Car.

3.5 Threads

The purpose of this program is to be able to specify a larger concurrent program using permission based separation logic. This program has two important classes that can be multithreaded. The MainCable, which can have multiple FeedingCables, and the class FeedingCable. There can be multiple FeedingCables,

The cables move separately from each other, and they have to interact when a Car needs to pass from the FeedingCable to the MainCable or the other way around. It acquires permission to variables from the other classes by using Locks, which are created in the class to which its locks hold. It waits for the permission by using the .lock method, which waits until it acquires the lock before returning. We chose for this because the methods are not supposed to hold the locks long, and the acquiring of the locks are necessary for the thread to continue.

The program can have a single MainCable and multiple FeedingCables, each one being a separate thread. There is no limit in the amount of FeedingCables, and every instantiation of the FeedingCable also means using a new thread, so the program can be scaled to an unlimited amount of threads. It is assumed that once a cable exists, it won't be deleted anymore, so the threads will not end unless the entire program will stop. To shutdown the program, the MainCable and FeedingCable have a shutdown method. The run method is a loop that checks every time if the Boolean running is still true, and if it is false, it stops the loop, and thereby ending the run method, and gracefully exiting the thread.

Since, as stated above, the program uses multiple threads, which do a non-trivial task and since the program is a model of real-life scenario that has safety critical aspects, since a ski-lift is a form of transportation of humans, which makes safety very important, we may conclude the answer to the first subquestion is that this program is a suitable case study to try to specify using permission based separation logic.

4. SPECIFICATION

After building the program, it needed to be specified. When specifying the program, we focused on the occurrence of data races. When trying to specify functional behavior, there were some problems as explained in Section 4.4. We started specifying the program by analyzing which variables would need which permissions.

In a multithreaded program, the amount of access a thread has on a reference varies between the different threads. Some references will only be accessed by one thread, while other references will be accessed by multiple threads. The specification of the references used by only one thread is fairly straightforward, any method that uses this reference can request full write permission. The references which are not final and which are accessed by multiple threads need more complicated solutions, for example by using locks. To see which references needed those solutions, we started by identifying the variables that are shared between threads. The parts that are using multiple threads are MainCable and FeedingCable.

We created a diagram showing which variables were used by which method, including the usage of how MainCable and FeedingCable uses each others methods. In this diagram, straight lines are write acces and method invocations, while blocked lines are read access. After creating the diagram, we colored the variables which are shared, where red means write acess, and yellow means read access.

Appendix A shows a diagram including the variables in both the MainCable and the FeedingCable. As shown in the diagram, we found that distanceBetweenCars and carsOnCable in the MainCable are shared variables that are only read from other threads, while buffer is a shared variable that requires write access from another thread. In FeedingCable distanceBetweenCars and carsOnCable requires read access from other threads, while buffer and Queue requires read and write access. It is shown that in both classes addCar is used from different threads and needs write-access to buffer. This means the addCar methods in both classes needed the most work.

4.1 Specifying the permissions

4.1.1 Non shared variables

When a variable is not shared between threads, the thread that holds it always has a permission of 100 on it, meaning it has write permsission on it. Methods that need that variable still needs to require a permission in the specification, on the variable, but this will not be a problem, since it always has permission. The permission on variables are with the thread that creates the variable. The constructor of a method does not require any permissions, but ensures all permissions on the object, since at that point they are in possession of that thread. The permissions can be taken by another thread by using a lock. As will be explained in the next section. An example used in the program is the method isBlocking, which checks based on two positions, if these positions are blocking legal positions to have cars on at the same time, based on the permitted distance cars minimally need to have. For this it needs the distanceBetweenCars variable, which contains the minimum distance two cars need to keep of each other.

```
/*@
 requires Perm (distanceBetweenCars,50) **
incomingPosition > 0  && carPosition > 0;
 ensures Perm (distanceBetweenCars,50);
@*/
private          boolean          isBlocking(int
incomingPosition, int carPosition)
```

Listing 2: example of requiring and ensuring permissions on a reference not shared with other threads.

4.1.2 Shared variables

When a variable is used in between different threads, a method is necessary to pass the permission between threads. In this program we used locks to specify this permission. The specification of the locks grant permission to the variables when locking, and releases the permission to the lock when unlocking. In this scenario, a method can't require and ensure a certain permission, since it is possible another thread also uses a method that requires the same permission. The specification of lock ensures a certain permission (100 for write locks, and a low number for read locks), and the specification of unlock requires the same permission, thereby releasing the permission the thread had of the reference. The specification of the Lock class and it's methods can be found in the document Formal Specifications for Java's Synchronisation Classes[3].

In listing 3 is an example of how locks are used in the

program. In the FeedingCable, the buffer and carsOnCable variables are shared. Since they both are dependent on each other,they are protected by the same lock. As seen in Listing 3, the lock gets the resource posLock, which handles the permission on both buffer and carsOnCable. It also gets passed the value true as the second parameter, which means it is a reentrant lock. After locking the permissionLock, it assumes permission over the variables, which it gives back after using unlock. In this snippet, carsOnCablePerm is a predicate to make sure it has permissions on the entire array, as will be shown in the next section.

```
//@resource posLock(frac p) = (Perm(buffer, p)
** Perm(carsOnCable,p) ** carsOnCablePerm(p)
);

private ReadWriteLock
/*@ <posLock,true> @*/positionLock = new
ReentrantReadWriteLock();/*@<posLock> @*/
/*@
  requires cablesPerm(50) ** i > 0;
  ensures cablesPerm(50) ** true ;
 @*/
private void moveCars(int i)
{
//Here there is no permission on buffer or
carsoncable
//Some code...
positionLock.writeLock().lock();
//Here there is write permission on
carsOnCable
carsOnCable[carID] = null;

positionLock.writeLock().unlock();
//Here there is no write permission anymore

}
```

Listing 3: An example of the positionLock.

Another lock we used is for the queue, as shown in Listing 4. The queueLock is also a readwrite lock. In the sample in Listing 4, we see a usage of the readlock. After calling queueLock.readLock.lock(), the thread will have permission to read the amount of people in the queue, but won't be allowed to write to it. As shown in the examples, both using a readLock and a writeLock are fairly straightforward, however, as will be shown in the next sections, there were some other technical problems that had to be solved to create the specification.

```
//@resource qLock(frac p) = (Perm(queue,
p));
private ReadWriteLock/*@ <qLock,true>
@*/queueLock = new
ReentrantReadWriteLock/*@<qLock> @*/();
/*@
  requires Perm(placeOfMainCable,100)**
Perm(cable,100) **  Perm(storage,100) **
position > 0;
  ensures  Perm(placeOfMainCable,100)**
Perm(cable,100) **  Perm(storage,100);
 @*/
private void moveCars(int position){
//more code
queueLock.readLock().lock();
if (storage.getCapacity() >
storage.Count() && car.getLoad() == 0 &&
queue > 0)
{
    storage.addCar(car);
    carsOnCable[i] = null;
}
queueLock.readLock().unlock();

//more code
}
```

Listing 4: An example of the queueLock

4.2 Dynamic structures

While building the application, we used dynamic structures like ArrayLists and Stacks in several classes. When specifying these it was found out that it is hard to specify the permissions for dynamic structures.

4.2.1 The problem

When specifying a permission on a dynamic structure, the permission only applies to the reference of the structure, not the references inside the structure. Listing 5 shows an example of a permission on an ArrayList. In this example, if another thread has access to one of the elements inside the ArrayList, this thread can still get permission on the ArrayList, but when trying to acces one of the variables inside, a data race can still occur. Originally, a List was used to specify which cars exist on the cable, as shown in Listing 5. In this example, the method isLegalEmptyPosition requires permission for carsOnCable, however, in this specification it does get permission for the items that are inside of the list.

```
private List<Car> carsOnCable = new
ArrayList<Car>();

/*@
requires Perm(buffer, 100) **
Perm(carsOnCable,100) **
Perm (distanceBetweenCars,50);
ensures  Perm(buffer, 100) **
Perm(carsOnCable,100) **
Perm ( distanceBetweenCars,50);
@*/
private boolean isLegalEmptyPosition()
```

Listing 5: Example for dynamic structures

4.2.2 Solution

A solution to specify a program with dynamic structures is to use predicates, as shown by M. Roo[7]. A downside of this method is that for every different dynamic type, another predicate solution is needed.

Since we used multiple dynamic structures, such as Hashmaps, stacks and arraylists, we looked for a different solution. The solution we used was to rewrite the dynamic structures to arrays. To still make the program manageable, we created some helper functions that fills and reads the data from the array. To convert the stack to an array, we used an array with Cars, and used an index. Every time a car is added the car is added to the position of the index, and the index is increased.

Changing the dictionary to an array was also quite straightforward. Since the maximum size of the array is known

at the instantiation, it doesn't need to be expanded. And since the dictionary was already used with integer based indexes, all the get(i) could just be replaced by [i].

The list was slightly less straightforward. Since Cars in the carsOnCable list can be added anytime, and Cars can be taken out in any order, Cars can be spread over the array, so when adding a new Car, a new place needs to be found. To handle this we created the helper method addCarToArray, which loops over the array and takes the first open spot. Since ordering doesn't matter, this can be done without a problem.

By using an array, it is possible to request permission on the separate entries in the array. Since we have methods that need access to entire arrays, we created helper methods to do so, as shown in Listing 6. It loops over every position in the array, and calls Perm on it. A usage of this method can be seen in Listing 3, where the positionLock uses this.

```
//@resource carsOnCablePerm(frac p) =
(\forall* int i ; 0 <= i && i <
carsOnCable.length ; Perm(carsOnCable[i],p) );
```

Listing 6: An example of a helper predicate to specify the permissions

Everywhere where a method needs access to the all the entries in the array, a similar predicate has been used. However, when only one place in the array will need to be edited, it only needs access to that specific location in the array. This has been used by the method getCar, which takes a car out of the storage and passes it over. Since the array is built as a stack, and the index points to the next free slot, it only needs permission to the position of index-1. Also, since it changes the index, it needs write permission on the index. This is shown in Listing 7.

```
/*@
  requires Perm(cars[index-1],100) **
Perm(index,100);
  ensures Perm(cars[index-1],100) **
Perm(index,100);
 @*/
public Car getCar()
{
    Car retval = null;
    if (index > 0)
    {
        retval = cars[--index];
    }
    return retval;
}
```

Listing 7: Example of the usage of only one position of the array

4.3 Specifying functional behavior

The specification that we written for this program focuses on data races. However, we also looked at specifying more of the behavior of the program. The problem that arises here, is that other threads can change variables before the method has been returned. For example, the moveCars methods in both the FeedingCable and the MainCable move the cars a specific amount of places around the cable. However, when we tried to specify this in a post-condition, we ran into a problem. Since the lock gets released before the method is finished, there can be changes to the Cars on the Cable, for example a new Car can be added, before the method returns, thereby invalidating the postcondition. Research has been done to verify functional behavior of a program [8], which uses histories of variable assignments to be able to reason about a program. This is a time consuming process, therefore we did not do this during this research.

4.4 Using VerCors

After specifying the program we also tried to run the program through VerCors, so it could also be verified. We started with the option *–passes=java*, so it would not try to verify the program, but it would only run through the specification. We found that the program would need severe changes to make VerCors even run through the program. We found that VerCors would not allow the using of packages, or the using of import. Also the usage of annotations, of which we used the @Override annotation, would not let VerCors run through the code. This means that for a program to be verified using VerCors after specifying the program using permission based separation logic, it would still need work to be able to run it through VerCors.

4.5 Specification overview

The second and third subquestions are what problems arise while specifying the program, and how the problems that arise can be solved.

We successfully managed so specify the ski-lift program using permission based separation logic and use reoccurring specification patterns. The problems we found were as stated in Section 4.3 and Section 4.4 to specify permissions on dynamic structures, and the specification of formal behavior. As shown in Section 4.3, it is possible to avoid dynamic structures, thereby avoiding this problem, or use predicates, as used in specifications of other programs.

We did not specify much functional behavior, although as showed in Section 4.4, there are methods to still make statements about the functional behavior [8].

5. RELATED WORK

Proof of concepts exists for both VerCors and VeriFast. The proof of concepts from VeriFast also includes more complex and larger examples. For example there are drivers and kernel modules that have been verified using VeriFast. The case studies that exist for VeriFast are more complex. An example of case studies made for VeriFast is a game server for rock-paper-scissors, and parts of the C runtime library.

For VerCors the existing programs are mostly small examples. These examples are useful to see how specific situations can be proved using VerCors, and have been useful in the creation of this program. This project was done to try the specification method of VerCors on larger projects than the small samples that already exist.

6. CONCLUSION

In this section we will first start by answering the subquestions, after which we will answer the main research question.

How can the proposed model be built in a concurrent way such that it serves as a good use case to be specified using permission based separation logic?

In this paper we showed a model of the ski-lift, which has more complexity than a simple example program, using multiple threads. Furthermore, because of the safety critical characteristics of the program it is very suitable as a case study for specifying it using permission based separation logic.

What problems arise when trying to specify the build ski-lift program using permission based separation logic so it can be used to eventually verify the program?
In this paper we showed two problems that risen, the specification of dynamic types and the specification of more specific behavior. We discussed a solution for the first problem, and pointed to a solution for the second problem.

How can the problems that arise be solved?
The problems we ran into with dynamic structures could be solved by changing the dynamic types to arrays. While that worked in this example, there could be cases that this solution would not be sufficient, for example when there are reasons to stick with the dynamic types, for example because of performance reasons.
The problem that the specification cannot be used to cover functional behavior has been solved by making the specification less specific and skipping the functional behavior specifications, but this solution is not preferable. Given a larger timeframe for the project, the solutions staded by M. Zaharieva-Stojamovski, M. Huisman and S. Blom [8] could be applied to fully specify the behavior of the ski-lift.

How difficult is it to specify a multithreaded model of a complex ski-lift using permission based separation logic?
In this paper we showed that is possible to specify a more complex program, although some modifications need to be made to the program in order to build a correct specification. Future attempts to specify a larger program can use the solutions used in this project, as well as the solutions we did not use, but did discuss for their project.

7. FUTURE WORK
In this paper we created a program and a specification that can be used to verify it using the VerCors toolset. However, we did not verify it. Future research could try to verify the program we build. Another recommendation for future work is to extend the specification to also cover the functional behavior of the program.

8. REFERENCES

[1] A. Amighi, S. Blom, S. Darabi, M. Huisman, W. Mostowski, and M. Zaharieva-Stojanovski 2013 Verification of Concurrent Systems with VerCors DOI=http://doi.acm.org/10.1007/978-3-319-07317-0_5

[2] A. Amighi, S. Blom, M. Huisman, M. Zaharieva-Stojanovski. 2012. The VerCors Project: Setting up basecamp. DOI= http://doi.acm.org/10.1145/2103776.2103785.

[3] A. Amighi, S. Blom, M. Huisman, W. Mostowski, M. Zaharieva-Stojanovski 2013 Formal Specifications for Java's Synchronisation Classes

[4] C.A.R Hoare 1969 An axiomatic basis for computer programming DOI=http://dx.doi.org/10.1145/363235.363259

[5]]B. Jacobs and F. Piessens 2011 Expressive modular fine-grained concurrency specification. In POPL, pages 271–282, 2011. DOI=http://dx.doi.org/10.1145/1926385.1926417

[6] Susan Owicki and David Gries 1976 Verifying properties of parallel programs: An axiomatic approach. Commun. ACM, 19(5):279–285, 1976 DOI= http://dx.doi.org/10.1145/360051.360224

[7] M. Roo 2014 Specifying Concurrent Programs: a Case Study in JML and Separation Logic

[8] M. Zaharieva-Stojanovski , M. Huisman, S. Blom 2014 Verifying Functional Behaviour of Concurrent Programs DOI=http://dx.doi.org/10.1145/2635631.2635849

[9] D. Zwet 2014 The code with the specification of the ski-lift http://fmt.cs.utwente.nl/education/bachelor/132/

APPENDIX

A. Diagram