Automatic verification of concurrent programs

Filip Wieladek

Master Project Report

Chair of Programming Methodology
Department of Computer Science
ETH Zurich

http://www.pm.inf.ethz.ch/

SS 2010

Supervised by:

Dr. Alexander Summers

Prof. Dr. Peter Müller

Chair of Programming Methodology

inf | Informatik
Computer Science

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Abstract

Proving the correctness of sequential programs has been a research issue for the past decades. Over time, a lot of approaches has been developed on proving sequential programs.

Over the last ten years computers have slowly stopped becoming faster and instead started to gain multiple execution units. This change triggers a need for concurrent programs and the verification of these programs.

This project looks at Chalice, an object based programming language with a compiler and verifier which is able to verify the absence of deadlocks and race conditions in concurrent programs, and tries to verify the Composite pattern in a concurrent setting.

Contents

1 Introduction **13**
1.1 Aim . . . 13
1.2 Motivation . . . 13
1.3 Overview . . . 15

2 Background **17**
2.1 Concurrency . . . 17
2.1.1 Threads . . . 17
2.1.2 Race conditions . . . 17
2.1.3 Dead locks . . . 19
2.2 Chalice . . . 21
2.2.1 Methods and functions . . . 22
2.2.2 Method Contracts . . . 22
2.2.3 Permission . . . 23
2.2.4 Predicates . . . 25
2.2.5 Invariants . . . 27
2.2.6 Lock Ordering . . . 29
2.2.7 Threads . . . 30
2.3 Composite Pattern . . . 31
2.3.1 Description . . . 31
2.3.2 Motivation . . . 31
2.3.3 Example . . . 32

3 Writing to the Composite **35**
3.1 Composite Instance . . . 35
3.2 Implementation . . . 36
3.2.1 Specifying the invariant . . . 36
3.2.2 Method Implementation . . . 39
3.3 Remarks . . . 41

4 Reading the Composite **45**
4.1 Potential Solutions . . . 45

4.1.1 Considerate Reasoning . . . 45
4.1.2 Multiple Lock types . . . 46
4.1.3 Object Groups . . . 48
4.2 Solution Proposal . . . 51
4.2.1 Prerequisites . . . 51
4.2.2 Solution . . . 57

5 Additional Work 63
5.1 Eclipse Plugin . . . 63
5.2 Finding Bugs . . . 65
5.2.1 Generated Chalice Code uses default package . . . 65
5.2.2 Nullpointer exception at function post condition . . . 65
5.2.3 Lockchange on return variables . . . 66
5.3 Interesting Chalice features . . . 67
5.3.1 Predicates and *mu* fields . . . 67
5.3.2 Destroying access permissions . . . 67

6 Conclusion 69

A A simple implementation of the Composite pattern 71

List of Figures

1.1 The percentage distribution of physical processors in home computers 14

2.1 UML diagram of the Composite pattern 32
2.2 An example of a Composite pattern instance 33

3.1 An example of a possible instance in the implementation 36
3.2 A sketch of a graphical user interface window 43
3.3 The composite tree representing the window from 3.2 43

5.1 A screen shot of Eclipse running the Chalice plugin 64

List of Listings

2.1 Naïve pseudo code implementation . . . 18
2.2 A thread-safe code implementation . . . 19
2.3 An implementation with deadlocks . . . 20
2.4 Simple Chalice example . . . 21
2.5 The sqrt method signature . . . 22
2.6 A simple counter with an increment method . . . 23
2.7 A simple complete counter with an increment method . . . 23
2.8 A simple class with contracts . . . 25
2.9 A simple class with predicates . . . 25
2.10 A simple example with monitor invariants . . . 28
2.11 A simple example with locking . . . 30
2.12 A simple example with forking . . . 30
3.1 The invariant part of the write only implementation . . . 36
3.2 The *correctTotal()* implementation . . . 37
3.3 The *valid* predicate . . . 37
3.4 The *addLeft()* method . . . 39
3.5 The *addTotal()* method . . . 40
4.1 A simple example which produces an error . . . 51
4.2 Pseudo code showing regular locking in Chalice . . . 52
4.3 Pseudo code showing read write locking in Chalice . . . 53
4.4 Pseudo code showing read write and upgradable read locking in Chalice . . . 54
4.5 Pseudo code showing how shadow permissions work . . . 56
4.6 Pseudo code without shadow permissions . . . 56
4.7 Extension of Composite code to hold the invariant for structures . . . 57
4.8 Extension of Composite code showing how to add a new composite object to the structure . . . 58
4.9 Pseduo code for the structure . . . 59
listings/full.chalice . . . 71

List of Tables

4.1 Resulting lock from calling *downgrade* or *upgrade* 55

Chapter 1

Introduction

1.1 Aim

The goal of this project is to extend current technologies and methodologies to support the verification of concurrent programs which feature complex software design patterns. The verification of concurrent programs includes verifying their correctness, with regards to the specification of the program, verifying the absence of deadlocks and verifying the absence of race conditions. The programming pattern which is to be verified is a concurrent version of the Composite pattern. The technology which will be used is the Chalice programming language[2].

1.2 Motivation

Writing correct sequential programs is hard. Writing correct concurrent programs is a lot harder, as concurrent programs have multiple threads which can possibly interact with each other. Additionally the resulting bugs may appear randomly, as the execution order of threads is non-deterministic, which makes debugging very difficult.

Figure 1.1 shows the adoption of multiple processors in home computers[3]. This data is taken from an online gaming platform called Steam, which performs monthly hardware surveys. In 2004 most home computers were still single processor machines. This started to change in 2006 when computers with two physical processors started to push out the single processor machines. This trend continued and in 2010 there are more multi processor machines than single processor machines in use. Unfortunately sequential programs, which only have a single thread of execution, cannot make use of this new technology. Therefore there is a growing need for concurrent programs, which are able to use the additional processors. Because concurrent programs are a lot more complex, successful verification of the correctness of these programs would help reduce overall costs of software development.

Currently the majority of software developers rely solely on testing when writing software. Unfor-

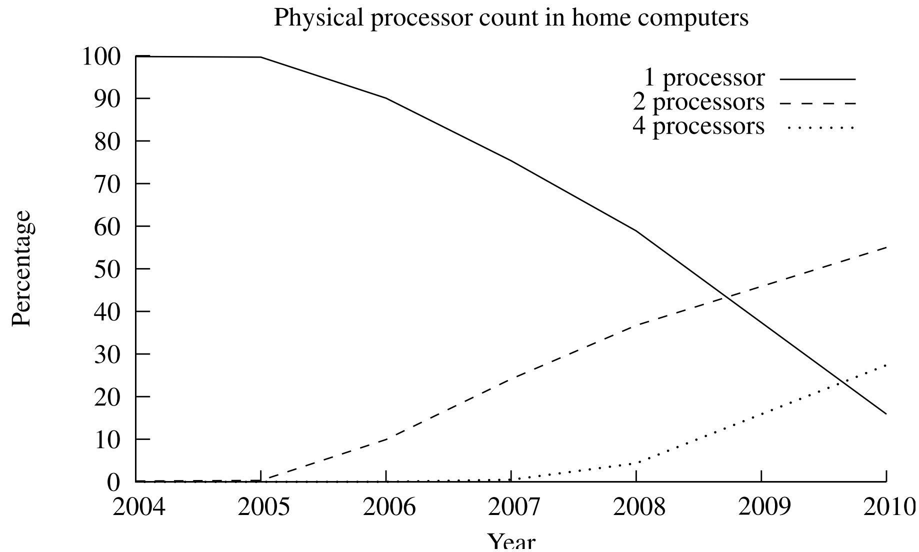

Figure 1.1: The percentage distribution of physical processors in home computers

tunately, testing can only ever prove the presence of bugs, but never the absence. Testing cannot check for all possible inputs as there might be potentially a very large number of combinations in a non-trivial program. Additionally, it does not help at all that testing usually needs to be done manually, whether it is writing unit tests cases or specifying the input classes, and in most cases it is monotonous and unrewarding. Furthermore, some bugs are very hard to find, especially when it comes to the area of concurrency. The most common bugs are dead locks and race conditions, both of which are very hard to find, because they may appear to at random and sometimes race conditions bugs are discovered at a much later stage than they actually happen.

Another use of verification is in environments where testing is simply not reliable enough due to the large number of possible inputs and the consequences of failure are very high. In the last years, drive by wire cars have become popular, especially among the premium class cars. These cars do not have any physical connections between the pedals and their respective mechanical hardware to execute the function of the pedals. The computer on the car has sensors which checks whether a break pedal was pressed and then engages the breaks. The software running on such a computer should rather be verified than tested as the software must execute the function in almost any condition. A software fault could potentially cost the life of the driver.

On the other hand, the Composite pattern is a challenging example of a software engineering pattern. The Composite pattern has been proposed as a verification challenge[1] was the 2008 challenge problem at the SAVCBS workshop. This makes it therefore even more interesting to verify in a concurrent setting.

1.3 Overview

The rest of the report is structured in the following way. Chapter 2 will discuss the background information which is needed to understand the topic and issues discussed in the report. The following chapter 3 will discuss the first attempt at verifying the Composite pattern, including the issues which were faced during the implemention. Section 4 describes the issues of extending the first attempt to handle a more complete Composite pattern, which includes the issues which were faced and a potential solution. Chapter 5 will provide information on additional work which was done during the project. This will include the description of bugs found in Chalice and the Eclipse plugin developed for the programming language. Finally, the last chapter 6 concludes the report, summarizing the work done during the project and listing the possible future work on the topic.

Chapter 2

Background

This chapter of the report provides the information which is required to understand the issues which were faced and the solutions proposed in this report. Section 2.1 briefly describes concurrency in general, describing threads, dead locks and race conditions. The following section 2.2 gives a brief introduction to the Chalice programming language, including the syntax, features, examples and usage of the language. Finally, the last section 2.3 provides a quick introduction to the composite pattern.

2.1 Concurrency

Concurrent programs are applications which have multiple threads. This section provides a brief overview of threads, race conditions and dead locks.

2.1.1 Threads

Threads are the basic building blocks of concurrent programs. A thread is a unit of execution which can share memory with other threads running in the same process. Threads are scheduled according to the scheduling policy of the underlying operating systems and the scheduling cannot be predicted from the application level. Additionally, each thread usually gets only a small time quantum for execution, which usually allows the thread to execute couple of commands before it is interrupted. Therefore the processor interleaves commands from all running threads. The same applies for computers with multiple processors, but on these machines the commands are actually executed concurrently.

2.1.2 Race conditions

Race conditions are bugs which may result from multiple threads accessing and writing the same memory locations at once. Consider the following pseudo code found in listing 2.1.

The code describes a simple bank account class which contains one variable called debit. The debit variable stores the amount of money currently in the account. Note, that this is an account which

```
1  class BankAccount{
2    //describes the amount of money on the bank account
3    debit : int
4
5    method deposit(amount : int){
6      debit = debit + amount
7    }
8
9    method withdraw(amount : int){
10     debit = debit − amount
11   }
12 }
```

Listing 2.1: Naïve pseudo code implementation

allows for infinite credit, therefore allowing for negative credit. It is a highly generous bank where one can always withdraw money. This piece of code works as expected in a sequential setting, incrementing the debit variable when the deposit method is called and also subtracting from the debit variable when the withdraw method is called.

The problem appears once these methods may be called concurrently from different threads. Imagine that this account is connected to two people, Bob and Eve, who both want to deposit money (at different ATM machines) at the same time. The account has initially three hundred Euros (which is represented by the debit variable). Bob places an additional fifty Euros into the ATM and Eve places an additional hundred Euros into the ATM. At the end of the transactions, they should have four hundred and fifty Euros in the bank account. Let's however consider the following execution sequence.

1. Thread Bob enters the deposit method call with an argument of 50
2. Thread Eve enters the deposit method call with an argument of 100
3. Thread Eve reads the value of debit which is 300
4. Thread Bob reads the value of debit which is 300
5. Thread Eve adds 100 to 300 (which is 400)
6. Thread Bob adds 50 to 300 (which is 350)
7. Thread Eve writes the result (400) to the debit variable
8. Thread Bob writes the result (350) to the debit variable
9. Thread Eve leaves the method body
10. Thread Bob leaves the method body

Clearly, one can see that the end result in the debit variable is only three hundred and fifty instead of four hundred and fifty. This might not be a huge problem since the bank is not losing any money, but gaining. Unfortunately (for the bank) this can work both ways and may also happen with the withdraw method call. This is a classic example of race conditions. A race condition is a bug which appears when two threads read and write to the same memory location at the same time. Operations such as writing and reading cannot be done atomically and there is always the danger of being interrupted with other commands being interleaved. This is even more likely when there are multiple processors in the machine. The biggest problem with race conditions is that they are very hard to detect with regular testing. The above example may fail only very rarely as the possibility of the sequence of commands interleaving is low.

One solution to this problem is mutual exclusion. Mutual exclusion, also called locking, guarantees that a given set of commands will never be executed concurrently but always sequentially, therefore removing the possibilities of race conditions. Locking is usually done over a monitor object, which depending on the implementation can be the object itself. Locking a monitor ensures that no other thread is able to lock that monitor until it is release. Consider the example in listing 2.2.

```
1  class BankAccount{
2    //describes the amount of money on the bank account
3    debit : int
4
5    method deposit(amount : int){
6      lock this
7      debit = debit + amount
8      unlock this
9    }
10
11   method withdraw(amount : int){
12     lock this
13     debit = debit − amount
14     unlock this
15   }
16 }
```

Listing 2.2: A thread-safe code implementation

This piece of code now adds two commands called *lock* and *unlock* which create a mutually exclusive section of code for one object. This means that no two threads can be between *lock* and *unlock* at the same time for the same object. Instead, if one thread enters the critical section (i.e. executes the *lock* command) then the other thread which arrives at that command needs to wait until the first thread calls *unlock*. Now, the sequence of commands resulting in a race condition is no longer possible and therefore this code avoids race conditions.

2.1.3 Dead locks

The previous section (2.1.2) described how race conditions can be avoided with the help of locking. Unfortunately, locking also comes with a potential danger called dead locks. Dead locks happen

when two threads each have locked a given resource which the other one requires. Consider the pseudo code shown in listing 2.3.

```
1  class BankAccount{
2    //describes the amount of money on the bank account
3    debit : int
4    owner : Person
5
6    method deposit(amount : int){
7      lock this
8      lock owner
9      debit = debit + amount
10     owner.deposited(amount)
11     unlock this
12     unlock owner
13   }
14
15   method withdraw(amount : int){
16     lock owner
17     if(owner is allowed to withdraw)
18     {
19       lock this
20       debit = debit − amount
21       owner.withdrew(amount)
22       unlock this
23     }
24     unlock owner
25   }
26 }
```

Listing 2.3: An implementation with deadlocks

This is an extended example of the bank account class. In this example every bank account has an associated person, which is described by the *owner* field. This bank account now only allows to withdraw money if the owner has the required permission from the bank (e.g. this might be a time deposit where the person needs to wait some time before they can withdraw the money from the bank). Now, consider that Bob and Eve (who are both registered to this account) try to deposit and withdraw money at the same time. The following sequence of events is possible.

1. Thread Bob enters the deposit method call
2. Thread Eve enters the withdraw method call
3. Thread Bob locks the bank account object
4. Thread Eve locks the owner object
5. Thread Bob tries to lock the owner object, but that is already locked by Eve
6. Bob waits for Eve to unlock the owner object

7. Eve checks if the owner is allowed to withdraw money (turns out to be true)

8. Eve tries to lock bank account object, but fails as its already locked

9. Eve waits on Bob to unlock the bank account object

In the described situation, it becomes apparent that the two threads will never finish because Eve waits on Bob to unlock the bank account object, but Bob can only unlock the object after it has locked the owner object which is held by Eve. Therefore, we have a dead lock and the program will never finish executing. The solution for this problem is quite simple as it is enough to switch the locking order of *this* and *owner* in the deposit method, such that the owner object is always locked first.

As with race conditions, dead locks do not happen every time the program is run, but rather depend on how the threads get scheduled. Therefore it is also very difficult to test for deadlocks.

2.2 Chalice

Chalice is a programming language built on top of the Boogie a verifier that can prove the correctness of concurrent programs. Chalice is able to verify programs with the help of method contracts, invariants, permissions and lock ordering. These will be explained in the sections which follow. For now consider the simple Chalice example found in listing 2.4.

```
1  class Math {
2    method sqrt(n: int) returns (res: int)
3      requires 0 <= n;
4      ensures square(res) <= n && n < square(res+1);
5    {
6      res := 0;
7      while (square(res + 1) <= n)
8        invariant square(res) <= n;
9      {
10       res := res + 1;
11     }
12   }
13
14   function square(n : int) : int
15   { n * n }
16 }
```

Listing 2.4: Simple Chalice example

The piece of code presents a rather simple example of a (incomplete) math class. The class features a simple function *square* which multiplies any value by itself. The class also features a method *sqrt* which computes the square root of a number *n* using a simple iterative approach. The result of the computation which is returned to the caller of the function is written to the *res* variable.

2.2.1 Methods and functions

Chalice differentiates between methods and functions. Syntactically they look very similar both allowing for contracts (see Section 2.2.2), however there is a significant difference between them. A Method body, in Chalice, are a statement (or a sequence of statements) whereas the body of a function is only an expression. Thus methods are in the group of statements and functions in the group of expressions. This is a subtle, yet very important difference.

Since functions can only contain expressions, they can never change any object state, because Chalice expressions are side-effect free. Functions can be thought of as a way of abstracting over expressions. Therefore functions are expressions, which also means that they can be used in other expressions. Methods on the other hand cannot be called in expressions because they may possibly alter the heap. Additionally, methods can have multiple return values, whereas functions are allowed to only return one value.

Methods are called using a special *call* keyword. Another important difference is that functions are not are inlined into the places where they are used and only the preconditions are checked at a function level. Functions in general do not use post-condtions as the Chalice verifier will be able to deduce them.

2.2.2 Method Contracts

Method contracts take the form of *requires* and *ensures* and are part of the method (or function) signature. They appear after the return declaration (if there is any) and before the method body. Listing 2.5 shows only the method declaration of the math class example.

```
1 method sqrt(n: int) returns (res: int)
2     requires 0 <= n;
3     ensures square(res) <= n && n < square(res+1);
4   {
5     //implementation here...
6   }
```

Listing 2.5: The sqrt method signature

The signature of the method defines that any thread calling the method needs to make sure that it does not give an argument which is negative. This is called the pre-condition and is denoted by the keyword *requires*. If the thread passes an argument which is negative or did not ensure that the variable passed down is never negative, the chalice verifier will complain and will produce an error message. Consider the following.

```
1 call math.sqrt(4); //OK
2 call math.sqrt(−3); //NOT OK
```

The post-condition is defined by the keyword *ensures*. The post-condition states which assumptions the thread can make after the method execution has finished. In the example of the sqrt method (defined in listing 2.4) the caller of the method can assume that the square of the value returned will never be bigger than the argument and that the argument will be smaller than the square of the result plus one.

Additionally post-conditions can make use of the *old(expression)* expression which denotes the value of the expression at the start of the method call. The listing 2.6 shows an example on how to use the *old(expression)* expression.

```
1  class Counter {
2    var value : int;
3    method increment()
4      ensures getValue() == old(getValue()) + 1
5    {
6      value := value + 1;
7    }
8    function getValue() : int
9    { value }
10 }
```

Listing 2.6: A simple counter with an increment method

Note that the listing 2.6 will not verify as some details have been omitted (see Section 2.2.3) to make the example easier to read. The example shows a counter class with a method which will increment the variable value every time the method is called. The value can be read by calling the getValue() function, therefore hiding the implementation details. The post-condition states that *getValue()* will return the old value of*getValue()* plus one after the method *increment()* is finished.

2.2.3 Permission

In the previous section (section 2.2.2) the counter class (listing 2.6) had some parts of the contract removed to simplify the example. The missing code from the contract are permissions. In Chalice permissions are used to define "access rights" to given fields, which means that a thread needs to have permissions to read and write a particular field. This is the main mechanism by which Chalice prevents race conditions.

Listing 2.7 is now an example of the counter class with all the permissions in place. This example will now successfully verify.

```
1  class Counter {
2    var value : int;
3    method increment()
4      requires acc(value, 100)
5      ensures acc(value, 100)
6      ensures getValue() == old(getValue()) + 1
```

```
7     {
8       value := value + 1;
9     }
10    function getValue() : int
11      requires rd(value)
12    { value }
13  }
```

Listing 2.7: A simple complete counter with an increment method

The contracts now define that any thread which wants to call the method needs to have full access permissions to the field value. This is done using the *acc(field)* expression. Permissions can be split into percentages ranging from zero to one hundred. Therefore, one can also mention half of the permissions to a memory location by using *acc(field, 50)*. Additionally one can have ε permission which is an arbitrary small yet positive value, which can be denoted using the *rd(field)* expression. Note, that a special access command is *acc(this.*,x)* which describes access permissions to all the fields in the object.

Chalice defines a clear set of rules on how field reads and writes may be performed. In particular, a thread needs to have at least ε permissions to a given field to be allowed to read it. This can be seen in the counter class (listing 2.7) where the function *getValue()* requires ε permission to the value field. Therefore, the *getValue()* function is allowed to read the value field.

On the other hand, a thread may only write to a particular field if it has one hundred percent permission to the field. Therefore the method *increment()* may read and write to the value field.

Note that in Chalice, one cannot forge permissions and the total is always one hundred percent. Therefore if a particular thread has acquired one hundred percent permission to a particular field, no other thread may be reading or writing to this memory location. Thus no race conditions may happen in such a situation. Similarly if a thread has only obtained a small amount of permissions, no other thread may gain one hundred percent permission to that field, which results in reads being safe from race conditions.

Any post or pre-condition which mentions fields requires permissions as well. Therefore the following example will not verify.

```
1   class InValid{
2     var x : int;
3     method m()
4       requires x > 0
5       ensures x > 0
6     {
7       //some implementation here
8     }
9
```

```
10  }
```

Without access permissions to the x variable the pre-condition may hold at the beginning of the contract, but there is a possibility that the valie of x may change before the body of *m()* is executed. This does not limit the programmer in practical way, because if the method has some pre-condition with regards to x it will most likely be using x and requiring the access permissions.

Any method which requires permissions inhales the permissions when it is called. If a method states in the pre-condition that it requires certain access to a field, it will not automatically return these permissions. These permissions have to be given back using the post-conditions. This is called exhaling of the permissions. Methods are not required to return permissions if they choose to implement some functionality. The most typical usage would be invariants which are described in section 2.2.5. Note, that functions automatically return the permissions.

2.2.4 Predicates

The previous section (2.2.2) described contracts and how they are used in Chalice. The examples shown in that section are correct, however they lack information hiding. The code has to reveal the internal implementation even if the method used functions instead of variable names in the contract. This can be seen in listing 2.8.

```
1   class NoEncapsulation{
2     var x : int
3
4     method m()
5       requires rd(x) && getX() > 0
6       ensures rd(x) && getX() > 0
7     {
8       //some implementation here
9     }
10    function getX() : int
11      requires rd(x)
12    { x }
13
14  }
```

Listing 2.8: A simple class with contracts

The method *m()* and the function *getX()* both expose the internal representation of x. The solution to this problem is the use of *predicates*. Predicates provide an abstraction over permissions as well as expressions describing conditions. Predicates (similar to functions) have a single expression for their body. The difference is that the expression may (and in most cases need to) contain access permissions. Any field mentioned in the predicate needs have its the correct access permissions. Listing 2.9 now shows an example with predicates.

```
1   class Encapsulation{
```

```
2     var x : int;
3
4     predicate valid
5     { acc(x) }
6
7     method Init()
8       requires acc(this.*)
9       ensures valid && getX() > 0
10    {
11      //some implementation here
12      x := 5
13      fold valid
14    }
15
16    method m()
17      requires valid && getX() > 0
18      ensures valid && getX() > 0
19    {
20      //some implementation here
21    }
22
23    function getX() : int
24      requires valid
25    { unfolding valid in x }
26  }
```

Listing 2.9: A simple class with predicates

In this example the methods and functions do not reveal the internal representation of the class. In fact, any programmer using this class does not need to be concerned with what valid actually means. After calling the *Init()* method call, valid can be assumed to be true.

Chalice provides two views of predicates. Each program point has either an *abstract* view (also called folded view) or a *concrete* view (also called unfolded view). The abstract view is independent of the implementation, whereas the concrete view expands the predicate and makes the contents accessible.

Unlike functions, whose implementation is always available at any program point, predicates need to be switched from one view to another manually, using the statements *fold* and *unfold*. To be able to fold a predicate, the expression of the body needs to hold. One can think of *fold q* as a method requiring the body of q and then returning q. In the case of the example provided in listing 2.9 one can think of the fold valid to be a method call with the following contract.

```
1  method fold_valid()
2    requires acc(x)
3    ensures valid
```

Unfolding of a predicate is the reverse operation. An *unfold q* requires q to hold and ensures that the expression defined in the predicate holds. Therefore the example provided in listing 2.9 one can think of the unfold valid to be a method call with the following contract.

```
1 method unfold_valid()
2   requires valid
3   ensures acc(x)
```

Additionally, Chalice adds support for a *unfolding x in y* expression, where x is a predicate and y is an expression. This allows the code to unfold any predicate temporarily in an expression. This is especially useful for functions.

2.2.5 Invariants

Chalice provides support for two kinds of invariants: loop invariants and monitor invariants. The loop invariant deals only with while loops whereas the monitor invariant is defined for an object.

Loop invariant

The loop invariant is a boolean expression which needs to hold at every loop entry and loop exit. The invariant expressions typically contain variables which change in the loop body. This is called successive approximation. In fact, most programming loops work by successive approximation.

The listing 2.4 on page 21 shows an example of a loop invariant. The invariant *square(res)* $<= n$ requires that at each loop entry, the square of res is never greater than the n value. What follows is that when the loop finishes the verifier can assume *square(res)* $<= n$ to be true. Additionally it can deduce that the negation of *square(res + 1)* $<= n$ (which is *square(res + 1)* $> n$) is also true, since the loop has exited. Therefore, the post condition of *sqrt* method can be successfully verified to hold.

Monitor invariant

Most programming languages which have support for contracts, also provide support for class invariants. It is important to note that Chalice does **not** use class invariants, but instead uses monitor invariants, which are different from the class invariants.

Class invariants typically define some expression which is assumed to be true of an object at any function or method entry, and then this boolean expression needs to be ensured at the very end of the method. Inside the method the invariant may be broken.

A monitor invariant works differently. A monitor invariant in Chalice is a boolean expression guaranteed to be true whenever an object is not locked by any thread. The main difference between a monitor invariant and a class invariant is the locations at which the expression can be assumed to

be true. The monitor invariant of an object can be assumed true right after the point the object was locked. Consequently any thread which releases a lock or shares an object needs to ensure that the invariant holds at that point. Therefore invariants can be thought of holding only when the object is shared (available for multiple threads to take).

Monitor invariants are also allowed to hold permissions. Similarly to the predicates, permissions which are in the invariant are acquired when the object gets locked and have to be given up when the object gets shared or released. Therefore the thread needs to hold the required access permissions to release/share a given object.

These differences have quite an impact on how one deals with invariants in Chalice. Monitor invariants do not need to hold in between method calls. More importantly, they do not need to hold at all if an object is never shared or if it gets never released.

```
1  class Car{
2    var wheels : int
3    invariant valid && getWheels() == 4
4
5    predicate valid
6    { acc(wheels, 100)}
7
8    function getWheels() : int
9      requires valid
10   { unfolding valid in wheels }
11
12   method Init()
13     requires acc(this.*)
14     ensures valid && getWheels() == 4
15   {
16     wheels := 4;
17     //valid needs to hold here
18     fold valid
19   }
20
21   method drive()
22     requires valid
23     ensures valid
24   { /*broom brooom...*/ }
25 }
26
27 class Main{
28   method main()
29   {
30     var car : Car := new Car;
31     car.wheels := 3;
32     //monitor invariant doesnt need to hole here...
33     call car.Init();
34     //monitor invariant needs to hold here
```

```
35        share car
36        acquire car
37        //monitor invariant holds here
38        unfold car.valid
39        car.wheels := 3
40        fold car.valid
41        call car.drive();
42        unfold car.valid
43        //monitor invariant does not need to hold here
44        car.wheels := 4
45        fold car.valid
46        //monitor invariant needs to hold here
47        release car
48      }
49    }
```

Listing 2.10: A simple example with monitor invariants

Listing 2.10 shows an example of monitor invariants. In this example, there is a class Car and a class Main. The Main class defines the entry point into the program with the method *main()*. The class Car defines a simple object which has 1 variable called wheels. This denotes the number of wheels of the car. A Car usually has 4 wheels (unless it is being serviced or it is a Scott Sociable). Therefore the car class defines an invariant which states the car should always have 4 wheels.

The main method shows however, that with the provided implementation it is still possible to drive the car even though the method sets the number of wheels to 3. Therefore, the drive method should either have a precondition with *getWheels() == 4* or *getWheels() == 4* should be included in the valid predicate. It should also be noted, that between the *share* and *acquire* statements, the method has zero access permissions to the *wheels* field. That permission is stored inside the predicate, which is stored in the monitor invariant.

2.2.6 Lock Ordering

As previously mentioned Chalice is able to prove the absence of dead locks. This is achieved with the help of a field called *mu*. Any object in Chalice has this special field. It defines the order in which the objects may be locked. Recall that dead locks usually occur when threads lock objects in different orders. Therefore, this *mu* field disallows locking objects in different orders.

When an object is created the *mu* field is automatically initialized to the special value *lockbottom*, which indicates that the object cannot be locked. When an object is shared, the mu field is written to. Therefore, sharing requires a hundred percent access rights to the mu field, otherwise the write would not be safe from race conditions (Section 2.1.2) (and could possibly lead to dead locks). It is not possible to write to the *mu* field directly, but rather with the *share* keyword. The default behavior of the share statement is *share x above maxlock*, where x is the object to be shared.

The special value *maxlock* defines the maximum *mu* value of all locked objects at that point. That means that one can only lock objects which have a *mu* value above the *maxlock* to avoid deadlocks. The full *share* command takes the form of *share x below y above z* where x, y and z are all objects to which the thread has at least read access to the *mu* fields of y and z. Optionally, y or z can be exchanged for maxlock. Listing 2.11 shows a simple example showing how objects are shared. Note, that the object *y* has a *mu* field which is above the *mu* field of the x object. Therefore, Chalice will complain about the order in which the method *m()* tries to lock the items. The correct order would be *acquire x; acquire y;*. This also means that any piece of code wanting to lock a particular object, requires at least read access to the object's *mu* field.

```
1  class Foo{
2  }
3
4  class Main{
5    method m()
6    {
7      var x : Foo := new Foo
8      var y : Foo := new Foo
9
10     share x
11     share y above x
12
13     acquire y;
14     acquire x; //NOT OK, x.mu is below y.mu
15   }
16 }
```

Listing 2.11: A simple example with locking

Chalice introduces a special operator << which is used to compare *mu* values. An expression *x.mu* << *y.mu* evaluates to true if and only if the value of the *mu* field of *x* is lower than that of *y*.

2.2.7 Threads

Chalice also provides support for the creation of threads. Without that functionality, all the previously mentioned complexity would not be of much use. Chalice can create threads with the use of the *fork* statement. The *fork* statement works very similarly to the *call* statement. The difference is that the *fork* statement does not return the return values of the method, but instead the statement returns a *token* which can be used to wait on the result of the method. That method is then executed in a newly created thread. Listing 2.12 an example where two threads are created both of which compute the sum of the arguments. Notice that the return value is not the actual result, but a *token* which can then be waited on by using the *join* command. This command returns the actual sum.

```
1  class Main{
```

```
2
3     method main()
4     {
5       var sum1 : int;
6       var sum2 : int;
7
8       fork thread1 := add(3, 4);
9       fork thread2 := add(4, 6);
10
11      join sum1 := thread1
12      join sum2 := thread2
13    }
14
15    method add(x : int, y : int) returns (sum : int)
16      ensures sum == x + y;
17    {
18      sum := x + y;
19    }
20  }
```

Listing 2.12: A simple example with forking

Similarly, if the method call requires permissions, these permissions will be consumed at the *fork* statement and if a method ensures permissions these will be first received at the *join* statement. Otherwise two threads would be able to acquire a hundred percent permission to memory locations.

2.3 Composite Pattern

The Composite pattern is a software design pattern, which is the research subject in this master thesis. This section describes the Composite pattern, motivation for usage and finally gives a small examples on how the composite pattern may be used.

2.3.1 Description

The pattern describes groups of objects which can be treated as a single instance of an object. Figure 2.1 shows a UML diagram of the Composite pattern. The pattern is made up of three classes: Component, Leaf, Composite. The Component is the actual abstraction, and the Leaf and Composite both inherit from the Component. The Leaf and Composite can only have 1 parent and the Composite may have multiple children. Therefore, the structure forms a tree.

2.3.2 Motivation

The main motivation behind using the composite pattern is to simplify the resulting code which is required to support a certain feature. The code does not need to differentiate between a group of objects and single instances neither does it need to differentiate between a composite or leaf.

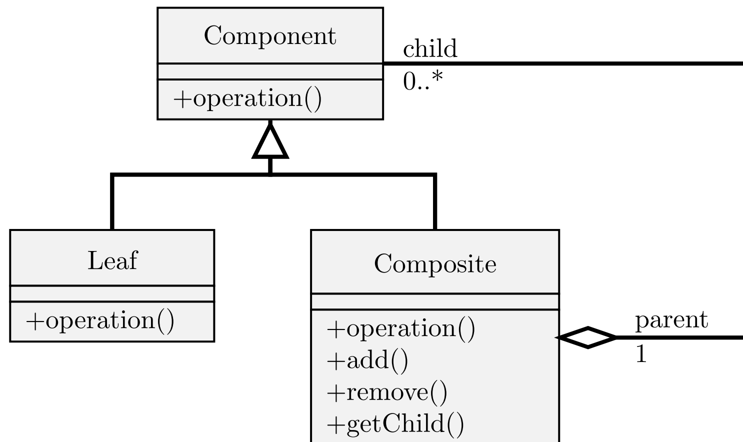

Figure 2.1: UML diagram of the Composite pattern

Consider the example of a windowing system. These are usually built using the Composite pattern. Any window component (e.g. panel, button, checkbox...) can be added to another. Therefore, one can add a button to a panel and the panel to the main window frame. Now if all of these components support a *setVisible()* method, then it is enough to call the method on any component to hide all the children of that component. Otherwise, one would have to iterate through all the children manually.

2.3.3 Example

Lets consider a slightly modified version of the Composite pattern. Chalice does not support inheritance, therefore one actually has to modify the pattern. The modified version of the pattern only uses one class, the Composite class. This class can both act as a leaf or as a composite and therefore there is no need for the component object.

Figure 2.2 shows an example of a Composite pattern instance. That instance of the Composite pattern represents a Dinner at a restaurant. Every object inside that pattern represents a meal and the associated calories. Additionally, one would want to define an invariant which states that the calories value of a given meal is the sum of all the calories values from all its children. Therefore, the root contains the calories for the whole tree. This is a property which should always hold.

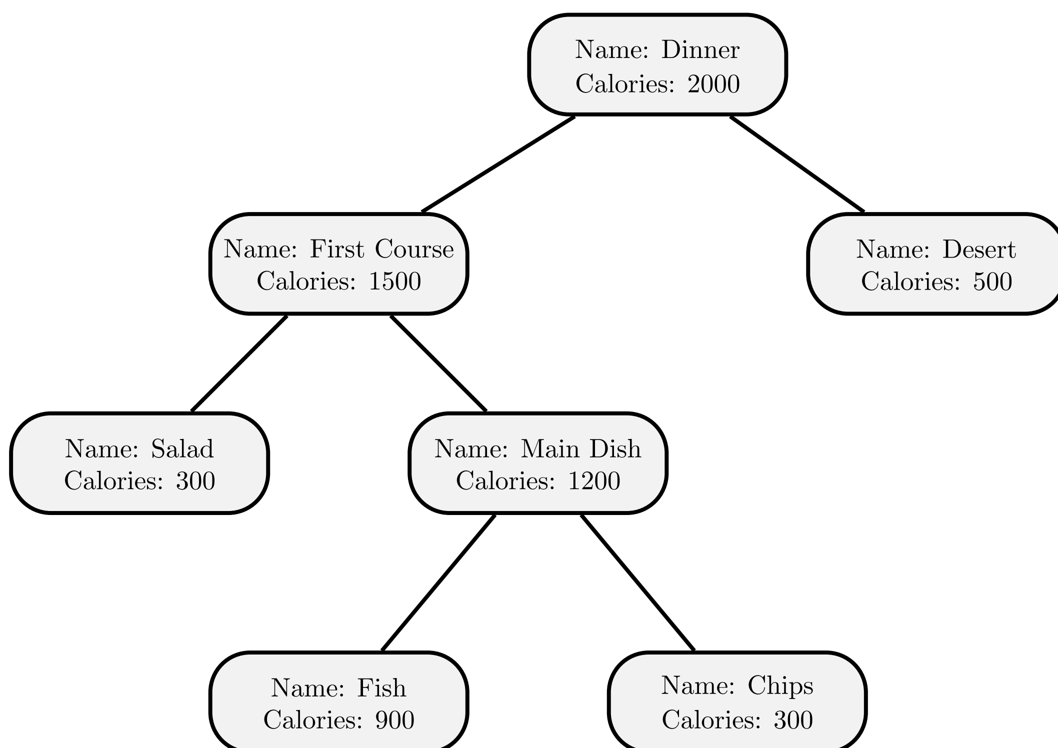

Figure 2.2: An example of a Composite pattern instance

Chapter 3

Writing to the Composite

This Chapter describes the a simple implementation of a Composite pattern in chalice. This implementation supports writing to the data structure, in the form of adding composite objects to the structure. This chapter begins with describing the instance of the Composite pattern which will be used, followed by the details of the implementation and finishes with issues this implementation faces.

3.1 Composite Instance

The implementation of the Composite pattern slightly differs from the one described in the background section (2.3). Chalice does not support inheritance, therefore, there are only objects of the same class used to implement the pattern. This does not limit the project in any way, as the interest is in verifying the invariant and not the pattern usage.

The Composite pattern had to be simplified to a binary tree, as Chalice did not support specifying access permissions to fields inside of arrays. Therefore, the implementation only allows for a "left" and "right" child. This again does not limit the implementation in any way as the simplification does not simplify the main problem, which is verifying that the invariant holds. On a side note, this bug has been fixed in lates Chalice build.

The invariant in which the project is interested in, is counting the total number of nodes in a sub-tree of the Composite pattern. Figure 3.1 shows an example of the Composite pattern which is implemented.

The diagram shows the total fields of each individual node. Each node has the total value which is the number of nodes in the subtree rooted at that particular node. For instance, the root node contains the number seven, and every leaf node contains the number seven.

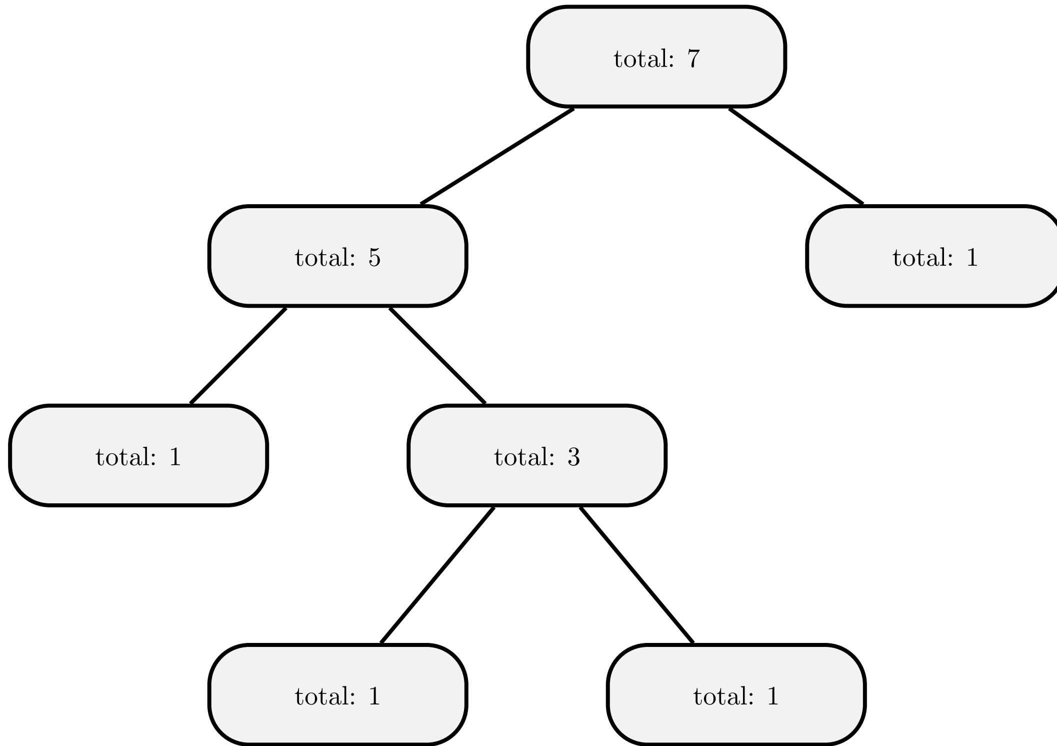

Figure 3.1: An example of a possible instance in the implementation

3.2 Implementation

Lets look at the implementation of the composite pattern. The full code for the implementation can be found in the appendix (see appendix A). This report describes the implementation step by step starting with the invariant(section 3.2.1), followed by the implementation of the most important method calls(section 3.2.2) and concludes with general comments about the approach taken(section 3.3).

3.2.1 Specifying the invariant

Listing 3.1 shows only the code responsible for the invariant in the class. The invariant for the Composite class is split into two, the *valid* predicate and the *correctTotal()* function. The *correctTotal()* function corresponds to the actual invariant of interest.

```
1  class Composite{
2
3    var parent : Composite;
4    var left   : Composite;
5    var right  : Composite;
6
7    var total  : int;
8    ghost var index : int;
```

```
9
10    invariant valid && correctTotal()
11
12    /*...*/
13  }
```

Listing 3.1: The invariant part of the write only implementation

Listing 3.2 shows the implementation of the *correctTotal()* function. This function requires *valid* to hold (the details of the predicate are shown in this section), and computes the correct total amount. This is achieved by checking whether the composite object has any children, summing up their total fields and adding one. This assumes that the total fields of the children satisfy the invariant, which can only be assumed true if the children are shared or are going to be released. In this implementation that does not pose a problem as every newly added object gets released.

```
1  function correctTotal() : bool
2    requires valid
3  {
4    unfolding valid in (total == (1 + ite(left == null, 0, left . total) + ite(right == null, 0,
         right . total)))
5  }
```

Listing 3.2: The *correctTotal()* implementation

Listing 3.3 shows the implementation of the *valid* predicate. The *valid* predicate contains the access permissions and structural invariants of the Composite pattern. The access permissions are set up in the following way. Every composite object has at least half access permission to all its own internal fields, which includes the parent, right, left, index, and total fields. The valid predicate can therefore read the fields and define additional restrictions.

```
1  predicate valid
2  {
3    acc(parent, 50) && acc(right, 50) && acc(index, 50) && acc(left,50) && acc(total, 50)
4
5    && (index == 0 || index == 1 || index == 2)
6    && (index == 0 ==> parent == null)
7    && ((index == 1 || index == 2) ==> parent != null )
8
9    && (left != null ==> acc(left.total, 50) && acc(left.parent, 50) && left.parent == this
10          && acc(left.index, 50) && left.index == 1)
11   && (right != null ==> acc(right.total, 50) && acc(right.parent, 50) && right.parent ==
         this
12          && acc(right.index, 50) && right.index == 2)
13   && (parent != null ==> ite(index == 1, acc(parent.left, 50), acc(parent.right, 50)))
14   && (parent != null ==> acc(mu,1) && acc(parent.mu,1) && this << parent)
15
16   && (index == 1 ==> parent.left == this)
17   && (index == 2 ==> parent.right == this)
18
```

```
19     && (left == null ==> acc(left, 50))
20     && (right == null ==> acc(right,50))
21     && (parent == null ==> acc(total, 50) && acc(parent, 50) && acc(index, 50))
22   }
```

Listing 3.3: The *valid* predicate

First of all, consider the index variable. This is a ghost variable, which is only used for verification purposes and never used in the actual implementation. This variable is required so that the composite object knows if its the left or right child in the parents fields. In general, if the parent had an array of objects as children, this variable would be the index the child reference is stored at. The index value can be used to hold the proper permissions and inspect the proper element.

The next expressions (line 9 up to 12) checks for children. For every child of the object, the composite object should hold the access permissions to at least the total, the parent and the index. The access to the total is required so that *correctTotal()* can read the value in the function. Read access to the parent and index variables are required so that the parent and index fields can be checked for consistency, i.e. the parent of the child should always point to *this* object and the index should also be correct (in this case, if the child is left then the index of the child should be one, otherwise two).

Expressions (on line 13 and 14) deal with the case when the object has a parent object. In here, the predicate states the access permissions to the parents *left* or *right* field. This is where the *index* ghost field is required. Without that field, the predicate would have to require both access permissions, to the *left* and to the *right* field. That might not seem to pose a problem, but it would not be possible to implement the locking as there is no locking order between the children, and therefore it would not be possible to lock another sibling object without locking the parent. The line 14 specifies that the parent object needs to have a higher *mu* field than the *this* object, so that one can lock up the tree.

The lines 16 to 17 are for consistency. They make sure that if the index is non-zero (i.e. the object has a parent), then the parent *left* field (in the case of index being one) or *right* field (in the case of index being two) has a value which points to the *this* object. Otherwise the structure would be inconsistent. Again, if there would no be an index value, then the expressions would have to look like the following.

```
1   (parent. left  == this  || parent.right  == this) && (parent.left != parent.right)
```

The major drawback with this approach is the fact that this code would not scale very well. In the case of an array of children, doing such an expression would be very cumbersome and less elegant than the index field. In contrast, the index field would work very well with arrays.

The last line deals with the case when either the *left*, *right* or *parent* fields are empty. In the case of the *left* or *right* field, the access permissions to *left* or *right* are completed. That means, that if there is no *left* object, then the *this* object may change this field because no object's invariant

depends on the *left* field. Similarly, if there is no parent object, then there is no object, which invariant depends on the *total*, *parent* and *index* field. Therefore, the valid predicate may contain one hundred percent access to these fields.

3.2.2 Method Implementation

The implementation is split into two methods which do most of the work. Listing 3.4 shows the *addLeft()* method along with its contracts. Note, that the implementation does not have an *addRight()* method as it's symmetric to the *addLeft()* method.

```
1   method addLeft(newLeft : Composite)
2     requires newLeft != null
3
4     requires holds(this)
5     requires holds(newLeft)
6
7     requires valid
8     requires newLeft.valid
9
10    requires !hasLeft()
11    requires newLeft.addable()
12
13    requires correctTotal()
14    requires newLeft.correctTotal()
15
16    requires acc(mu,1)
17    requires acc(newLeft.mu,1)
18    requires newLeft << this
19    requires maxlock == this.mu
20
21    lockchange this
22    lockchange newLeft
23  {
24    unfold valid
25    unfold newLeft.valid
26     left := newLeft;
27     left .parent := this;
28     left .index := 1
29    fold newLeft.valid
30    fold valid
31    call addTotal(unfolding valid in newLeft.total)
32  }
```

Listing 3.4: The *addLeft()* method

The method requires for both, the current object and the *newLeft* (the object which is to be added) to be locked. In this implementation it is not enough to have the valid predicate hold, because this method will be releasing the locks later on in the code. Since the objects are locked, the method also requires the valid predicates to hold and also requires that the total fields are correct

by requiring *correctTotal()* on both objects. This is a consequence of monitor invariants, which are not required to hold in between method calls (in contrast to class invariants. See section(2.2.5)).

The lines 10 and 11 mention the actual pre-conditions for the logic of the method. The method requires that the current object does not have a left child and similarly, that the object which is to be added does not have a parent. Otherwise the method would have to deal with current objects child.

The last lines of the contract (lines 16 to 22) deal with lock ordering and lock changes. In section 3.2.1 , the *valid* predicate was described. The predicate requires access to the current object's *mu* field as long as there is a parent. This is important so that the method described later can acquire locks on parent objects. Therefore the method also needs to require access permissions to the *mu* fields. There is also a requirement of newLeft being lower in the lock order than the current object, which is required for the *valid* predicate to hold. Finally, the last two lines indicate that the locks of *this* and *newLeft* might change in the method.

The method body of the *addLeft()* is straightforward. The *valid* predicates are unfolded, the correct values are set and then the predicates are folded again. Note, that at that point, it is only possible to release the *newLeft* object, as it's *correctTotal()* is satisfied. The current object has a *total* field which does not satisfy the *correctTotal()* expression. Consequently, the *addTotal()* method is called to fix the local invariant.

Listing 3.5 shows the implementation of the *addTotal()* method. Note that this method is meant for the class' internal use, therefore the requirements may rely on internal fields. The method requires that the current object is locked and that the object is in a consistent state, which is achieved by requiring the *valid* predicate. It is also important to notice that the method can only be called if the maxlock is below the current object's *parent.mu* field, since this method will have to lock the *parent* object if such one exists. Line 6 requires that the *total* field contains a value which is smaller than the value described by the invariant. In particular, the value needs to be smaller by the *amount* argument.

```
1  method addTotal(amount : int)
2    requires holds(this)
3    requires valid
4
5    requires unfolding valid in (parent != null ==> maxlock << parent.mu)
6    requires (unfolding valid in total) == correctTotalAmount() − amount
7    lockchange this
8  {
9    var current : Composite := this
10   var parent : Composite;
11   while(current != null)
12     invariant current != null ==> holds(current)
13     invariant current != null ==> current.valid
14     invariant current != null ==>
```

```
15                  unfolding current.valid in (current.parent != null ==> maxlock << current.
                        parent.mu)
16        invariant current != null ==>
17                  (unfolding current.valid in current.total) == current.correctTotalAmount() −
                        amount
18      {
19        unfold current.valid
20        parent := current.parent;
21        if(parent != null){
22          acquire parent
23          unfold parent.valid
24        }
25        current.total := current.total + amount;
26        fold current.valid
27        release current
28
29        current := parent
30        if(parent != null){
31          fold current.valid
32        }
33      }
34    }
```

Listing 3.5: The *addTotal()* method

The implementation of the *addTotal()* method is entirely in the while loop. Recall the *valid* predicate from listing 3.3 from page 37. Notice that any object which has a parent, only has fifty percent access to the *total* field and the rest of the access is held by the parent object. Therefore, before the thread can change the *total* field, it needs to lock the parent object, if such one exists. Then the current object can be released because it's invariant has been fixed by correcting the *total* field (line 25). At that point, the parent object has the invariant broken, because the child has a new value in the *total* value. The loop will also fix the parents invariant by repeating the process until the loop has reached the root object (e.g. a node with no parent object).

This implementation is also known as *hand over hand locking*. In that solution, the thread has only ever two objects locked, the object which has it changes done and the object which invariant depends on these changes. Such an implementation allows for a high degree of concurrency.

3.3 Remarks

The implementation described in this section is correct, yet rather incomplete. One problem with that particular solution is that it is impossible to remove children objects from parents, when having the parent object locked and the children objects not locked. For a method which would remove children to be useful, it would have to support removing children without knowing the children beforehand. The problem in this implementation is that the code can only ever lock up the tree, but not down the tree. To remove a node from the tree, both objects (the parent node, and the node to be removed) need to be locked. Since the client code cannot know the child object

before locking the parent object, it cannot lock the child object before the parent object. It is neither allowed to first lock the parent and then the child as the locking order would be incorrect and the resulting implementation could deadlock.

On the other hand, removing a parent from a composite object is fine, but was not implemented. The implementation is very analogue to the add implementation, but instead of giving it a positive amount, the argument to the *addTotal* method would be negative. This is only possible since the only other object relying on the current objects parent field is the parent.

It should also be noted that this implementation has a high degree of concurrency due to the hand over hand locking implementation. Composite objects may be added (and removed) to the structure concurrently, with these additions to be happening at the same time.

The implementation stumbles on another problem which involves contracts. The *addLeft()* method requires that the current object is already locked and therefore is able to mention any preconditions on the objects, such as having an available slot for the left object. However, the hand over hand locking released the lock before the method finishes, and therefore it has zero access permissions to any of the objects field. The result of this is that the method cannot ensure anything as the object might have already changed. This is a huge disadvantage as the code inside the method cannot be verified according to the method contract. One could also swap the implementation of the *addLeft()* method for a single line which releases both objects and the code would still verify.

The biggest problem of the implementation is that the structure cannot ensure a consistent reading of all the values in the composite tree. Consider the case of a graphical user interface (in short GUI) renderer. A renderer is a piece of software which visualizes a particular gui window on the screen by drawing each element inside the window. The gui window can be represented using the Composite pattern where every element, such as a panel or button, is a composite object. The renderer may want to query the sizes required for particular elements in the composite structure to know how much space to reserve for a given element.

Now the problem with the provided implementation is that the gui renderer will not be able to read the sizes from a consistent state. It may lock every object one by one (and only ever holding the lock to one element). This approach will result in unexpected behavior as other threads are allowed to modify the tree concurrently while the renderer is reading the objects in the composite tree.

Figure 3.2 a simple sketch of an example window, which confirms if the user wants to a quit or not and figure 3.3 shows the corresponding representation in composite form. Additionally, every element features a size field which represents the required size of that component taking into consideration all of the children element sizes. Notice that the elements have a very similar behavior as the code implemented in section 3.2.

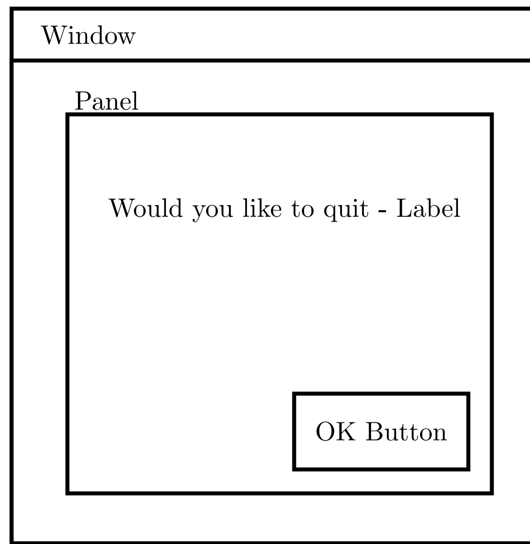

Figure 3.2: A sketch of a graphical user interface window

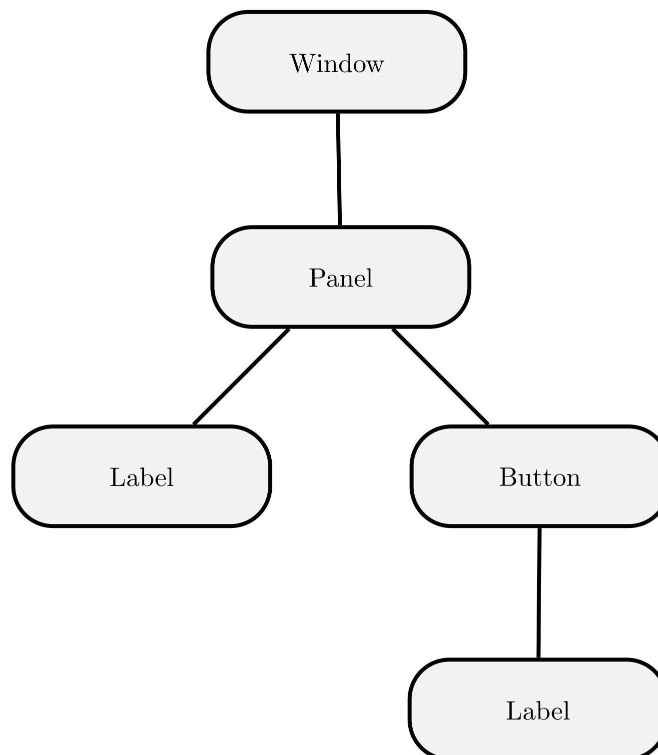

Figure 3.3: The composite tree representing the window from 3.2

Consider now that the gui renderer wants to draw the window on the screen. The first action would be to lock the top level element (which represents the window) and ask for its size. Now the renderer needs to make a copy of the list of all the children in the window and unlock the window object (since all children have a smaller *mu* field than the window element). Before the gui renderer is able to lock the panel element, another thread decided to add another button which allows the user of the program to cancel the quitting of the program. It is likely that this now increased the size of the panel as it received another component. The problem here will be the that the write may propagate now upwards before the gui renderer is able to lock the panel element. Imagine that the gui renderer locks the panel element after the write has completed. Now the panel seems to require more space than the window anticipated and might not fit inside the window anymore. This problem is attempted to be solved in the next chapter.

Chapter 4

Reading the Composite

The Chapter 3 described a simple and incomplete solution to verifying the composite pattern. It was shown that the Composite pattern implementation also requires a way to keep a steady state. This Chapter will describe ideas on potential solutions, prerequisites for a proposed solution and finally describe that proposed solution.

4.1 Potential Solutions

During the project, there were multiple ideas taken into consideration which were supposed to help in the successful verification of the Composite pattern with respect to reading a consistent state. This section describes the most important ideas.

4.1.1 Considerate Reasoning

The first idea was to look into into Considerate Reasoning[4], which is able to verify the Composite pattern in a sequential setting. It would have seemed that the approach taken by Considerate Reasoning could also be applied in a concurrent setting.

In short, Considerate reasoning is a technique for verifying sequential programs in which the verifier knows which fields (and their updates) break the invariant of which objects. This is useful, as the verifier can then check whether an object invariant has been reestablished. Additionally, Considerate Reasoning allows for temporarily broken invariants, which allows method to mention broken invariants and allows them to fix these invariants.

Unfortunately, Chalice would not benefit at all from using Considerate Reasoning. As a matter of fact, Chalice already uses a very similar approach. In Considerate Reasoning, the dependencies between fields and object invariants are implicitly defined. Chalice on the other hand has the dependencies defined explicitly in the form of permissions. If a particular invariant depends on another object and its field, then the invariant has to have at least read access to that particular field of that object. Therefore, Chalice does not benefit from Considerate Reasoning and was not

used in the project.

4.1.2 Multiple Lock types

Chalice only supports for locking objects in a single way which acquires all the access permissions and predicates stored in the monitor. The exception to this is the read/write lock in Chalice which is not completely implemented. The introduction of multiple locks would enable a higher degree of concurrency and possibly enable the implementation of read and write locks on the Chalice level.

The main idea behind multiple lock types is the following. Every object would define its invariant as it does right now in Chalice. Additionally, classes would be allowed to define lock types. These lock types would then define a subset of the monitor invariants permissions. The actual boolean expressions could be assumed at the time of acquiring the lock and would have to be ensured at release time. Consider the following example.

```
1
2  class Simple{
3
4    var x : int;
5
6    invariant acc(x, 100) && x > 0
7
8    locktype write: acc(x,100)
9    locktype read: rd(x)
10 }
```

This simple example defines a class Simple with a single integer memory location called x. The invariant states that the monitor holds a hundred percent access to the x field and that the x field should always be bigger than zero when the object is in a shared and unlocked state. Additionally, it defines two lock types which *write* and *read*. According to that specification, only one thread will be able to lock the write lock and all the other threads would have to wait until that lock is released. Similarly, there may be multiple threads with a read lock, but no thread with a write lock at any given time.

Every memory location which is mentioned in the invariant, requires to be in every definition of every locktype. This is a consequence of the fact that the invariant needs to be ensured when the object gets released. Imagine a class where the specification looks as follows:

```
1
2  class Simple{
3
4    var x : int;
5    var y : int;
6
7    invariant acc(x, 100) && acc(y,100) && x != y
8
9    locktype write_x: acc(x,100)
```

```
10     locktype write_y: acc(y,100)
11   }
```

In this example, when a thread acquired the *write_x* lock there would be no way of determining the if the boolean expression of *x != y* still holds. Therefore, in most cases the locks would be exclusive to one another.

This idea would not require a lot of modifications to the current Chalice compiler and verifier. The locks would be treated similarly as they are treated in the current implementation, with the only difference being the permissions the thread receives after acquiring a lock. The verifier would actually not have to deal with the number of threads which may lock a particular lock type at any given moment because the verifier is based on access permissions and does not care about other threads (which may have the same access permissions). As long as the thread has a fraction of permissions to certain memory locations, no other thread will be able to modify these locations.

This idea was not developed further as it suffers from couple of problems which make the solution undesirable. Firstly, it would be of an advantage if threads was able to upgrade the lock types to either full locks or different types of locks. This approach would unfortunately lead to deadlocks, as there is the possibility of two threads wanting to perform the upgrade simultaneously. This would result in both thread having a fraction of permissions which the other thread requires to get full access permissions.

These lock types would be mutually exclusive for most practical purposes, e.g. if there are threads which have obtained one of these lock types, no other thread may acquire the different lock type until all threads are done. This may not be clear at first, but becomes apparent upon further inspection. For lock types to actually make sense, the permissions in the types needs to vary, otherwise if the permissions are the same, then there is no point in having multiple lock types. If the permissions vary (e.g. one lock type provides total access to a particular memory location) then no other lock type may be locked at that time. In reality, the problem is even worse, as only one thread will be able to lock on the objects, and the additional complexity of this idea would not provide any benefit at all to Chalice.

There is also the issue on how to handle predicates in such a setting. Predicates actually hide the details on the permissions. Therefore, there would have to be some kind of mechanism which would provide a way to infer if a predicate contains a subset of permissions of another predicate.

With all the issues identified for this idea, it was rather clear that there is little merit in implementing such a feature into Chalice, as the implementation would not provide any more flexibility or concurrency for Chalice.

4.1.3 Object Groups

Since the Composite structure deals with a number of objects, it makes sense to group these objects into one structure which could restrict access to the objects or change properties of the whole group of objects without dealing with every single instance inside the group. There are multiple issues to consider for such a setting. The first issue deals with ensuring that the structure is deadlock free and the second issue is how a particular representation helps in verifying the Composite pattern. It turns out that this is not as straightforward as it would seem.

An object group would require to support at least two different modes of operation. The first mode of operation would be the write mode and the second mode would be the read mode. These two modes would be incompatible with each other, e.g. if an object group is in read mode and threads are currently accessing the objects inside the group, another thread which wants to set the group into write mode would have to wait for all threads to finish reading objects in that group. According to these modes different operations should be possible on the objects. The following sub sections deal with possible structure implementations.

Owners for every object

The simplest way of representing the structure is to enforce an owner field, which points to the structure the object belongs to. Additionally, every object requires every neighbor object to have the same owner. In the case of the Composite pattern, every object would require the parent object to have the same owner as the current object, so that one can ensure a consistent state when the read mode is active.

This results in a problem as suddenly it is not possible to add new objects to the composite structure. Consider the following piece of code.

```
1
2  class Composite{
3
4    var total : int;
5
6
7    predicate valid{
8      /* all old invariants here */
9      && acc(this.owner, 50)
10
11     //assumes at least rd(parent)
12     && (parent == null ==> acc(this.owner, 50))
13
14     //assumes at least rd( left ) present
15     && (left != null ==> acc(left.owner, 50) && left.owner == this.owner)
16
17     //assumes at least rd(right)
18     && (right != null ==> acc(right.owner, 50) && right.owner == this.owner
19   }
```

```
20  }
```

This could be supplementary to the implementation provided in chapter 3. Notice that in this case it is easier to enforce the condition that the child object requires the same owner as the current object, then requiring that the current object has the same owner object as the parent. Otherwise, the permissions would be hard to set up as every child would require access to a field in the parent object. In a binary tree that would not be that difficult, but for a general m-tree that could get very difficult to set up.

Now recall the *addLeft()* method from section 3.2.2. To ensure that the method does not break the parent invariant, the method has to change the *owner* field of the current object to the *owner* field of the parent object. Consider the following piece of code.

```
1   class Composite{
2
3    /* ... */
4
5    method addLeft(newLeft : Composite)
6    /*contract here*/
7    {
8      unfold valid
9      unfold newLeft.valid
10      left  := newLeft;
11      left .parent := this;
12      left .index := 1
13      left .owner := owner
14
15     //At this point newLeft.valid is not satisfied !
16     fold newLeft.valid
17     fold valid
18     release newLeft;
19     call addTotal(unfolding valid in newLeft.total)
20   }
21  }
```

The *fold newLeft.valid* will not be satisfied as the children objects might have different owner objects. The code at that point cannot lock the children objects anymore because these have a lower *mu* field than the *maxlock*. Therefore it is impossible to add new objects when having such a structure defined.

This implementation would also suffer from deadlocks. Consider two threads (called Bob and Eve for readability) who want to access the data structure. Bob wants to read from the structure and Eve wants to write to the structure. None of them actually know to which structure the objects they have references to belong. Assume that both Bob and Eve have references to different objects from the same structure. Bob and Eve both lock their object to acquire access to the *owner* field of the object. Eve manages to change the mode to write mode and continues with the write. Now if Bob and Eve are lucky, they will be able to finish the execution. On the other hand, it might

happen that Bob who wants to read the structure from a particular object, holds the reference to a parent object of Eve's object. Bob will try to set the mode of the structure to read mode and will have to wait until Eve is finished. Eve will be doing the update until it tries to lock Bobs object, which is held by Bob. Both threads are now deadlocked. This happens because the locking order is not satisfied. In general the owner structure should have a lower *mu* field than any object inside the structure.

Only Root has owner

Another way of representing the object groups would be to keep the owner in only the root object. This already assumes that the structure is a tree structure with a single root which can hold the structure field. For the Composite pattern this approach can be taken as the Composite pattern forms a tree structure.

In such an implementation, there is no problem with keeping all the owners consistent, as there is only one structure field in all the objects.

This approach, similar to the others, also has it's issues. The first issue is that its not clear how objects could ensure locally that they are indeed in the same object group. One possible solution would be ensuring that the root of the tree for the current object is exactly the same as the root for the parent object. It might be possible to define a function *rootOf()* which recursively finds the root of the a particular object. Such a function could be defined in the following way.

```
1  class Composite{
2     /*...*/
3    function rootOf() : Composite
4    /* contract */
5    {
6      ite(parent == null, this, parent.rootOf())
7    }
8  }
```

This function would require a complex contract and would also require all the objects to be parent objects to be locked. If someone were to write a contract which would look state *parent != null ==> (this.rootOf() == parent.rootOf())* Chalice would not be able to compute the statement without having all the permissions. That statement can easily be shown to be true by expanding the first function which would result in *parent != null ==> ((ite(parent == null, this, parent.rootOf())) == parent.rootOf())* . Since parent is not null in the implication, the expression can further be simplified to *parent != null ==> parent.rootOf() == parent.rootOf()* which is trivially true without even evaluating the *rootOf()* function.

The above problem could be avoided by possibly adding an additional rule for the verifier to assume the expression to be true at all times. Although that might work, the problem of the locking order still remains. To reach the root object one has to lock and keep the lock of all the

objects which are on the path to the root. Firstly, this is very inefficient, in the worst case, a thread might have to lock a large number of objects just to read two values from the tree in a consistent state. More importantly, this approach still does not solve the locking order issue. This technique still requires the actual object to be locked first in order to gain access to the correct structure object. As it was shown before, the actual structure needs to be set to a given mode before any operations on the actual objects are done.

4.2 Solution Proposal

This section of the report proposes a potential solution for verifying the Composite pattern, which supports reading and writing to the structure, with the additional support for setting the structure in a read or write state. The section starts with describing all the features required to support the solution but which are not currently present in Chalice and then continues with the actual proposal of a solution and finishes with remarks about the solution.

4.2.1 Prerequisites

Specifying Permissions over arrays

Chalice allows the handling of arrays and supports expressions specifying conditions over items in arrays. Unfortunately, at the time of the writing it was not possible to specify any access permissions over objects which were stored in arrays. Consider the example shown in listing 4.1.

```
1  class Test{
2    var tests : seq<Test>;
3    var total : int;
4
5    invariant acc(tests, 100);
6    invariant acc(total, 50);
7
8    //Problem 1
9    invariant forall { i in [0: size()−1]; acc(at(i).total,50) } //acc not expected here exception
10
11   //Problem 2
12   invariant forall { i in [0: size()−1]; at(i).pre } //assertion failed
13
14   function at(loc : int) : Test
15     requires acc(tests);
16     requires loc >= 0 && loc < size();
17   {
18     tests[loc]
19   }
20
21
22   function size() : int
23     requires acc(tests);
24     ensures result >= 0;
```

```
25      {
26          | tests |
27      }
28
29      predicate pre
30      { acc(total, 50) }
31  }
```

Listing 4.1: A simple example which produces an error

Line 9 and 12 both contain invariants which specify access permissions over a sequence of objects. Unfortunately, both these lines will produce an error in the Chalice compilation. The error produced is an internal exception which indicates this is a bug in the current implementation.

Specifying access permissions over objects in the array seems to be a very useful, for multiple purposes. Firstly, it is not possible to implement the Composite pattern which allows for multiple children without the ability to specify the access permissions to the children. Additionally, this would very useful for providing access to multiple object fields without having to lock these objects separately.

As a matter of fact, this bug was recently fixed and now Chalice is able to specify permissions overs objects in arrays.

Read-write Lock

Chalice currently only fully supports one type of locks on a monitor. This is a regular lock which provides full access to the permissions which are held by the monitor. The monitor can only be held by one thread at one given time.

It would be of big advantage if Chalice supported read/write locks. A read/write lock would allow any monitor to be locked in at least two state. The read state would only provide partial (e.g. ε) permission to all the permission specified in the monitor invariant. Additionally, since only partial permissions are used from the monitor invariant, that would imply that more than one thread is allowed to acquire a read lock on an object.

A write lock on the other hand would behave exactly the same as the locks behave right now. Once a write lock is acquired on an object full access permissions, as defined in the monitor invariants, are granted. In contrast to read locks, no other thread may acquire the object in any state when another thread has acquired the write lock

Ideally, the read/write lock has support for upgrading a held read lock into a write lock. This adds a higher degree of concurrency in certain situations. Consider the pseudo code shown in listing 4.2.

```
1  method doWork()
```

```
2  {
3    acquire this
4
5    if(this.x > 0){
6      this.x = 0;
7    }
8
9    /* Do some work here which requires *
10    * only read access to the field of x */
11
12   release this
13 }
```

Listing 4.2: Pseudo code showing regular locking in Chalice

This example shows how locks can currently be used in Chalice. The method *doWork()* first locks the current object and then changes the value of x to zero if x is greater than zero. The method then does some potentially time consuming work (possibly locking other objects) and only ever reads memory locations from the current object. Finally the method releases the lock.

The pseudo code shown in listing 4.2 is sound, but in general it does not allow for as much concurrency as it would seem to be possible. Note, that the only place which requires full access to the current object is line 6. All the other computation just requires read access to the x field.

Consider now the pseudo code shown in listing 4.3. This listing shows a modified example of the locking. Instead of using regular locks, one is now allowed to use *acquire.read* and *acquire.write* to indicate which lock one requests for the monitor

```
1  method doWork()
2  {
3    acquire.read this
4
5    if(this.x > 0){
6      acquire.write this
7      this.x = 0;
8      release.write this
9    }
10
11   /* Do some work here which requires *
12    * only read access to the field of x */
13
14   release this
15 }
```

Listing 4.3: Pseudo code showing read write locking in Chalice

In such a setting, it becomes apparent that there is higher degree of concurrency, since now multiple threads are allowed into the block which takes up a longer amount of time, since that block only requires partial permissions instead of full permissions. Also, keep in mind that the write

permissions are only ever needed when the value in the x field is bigger than zero. Depending on the application, this might actually happen very rarely, therefore reducing the degree of concurrency.

The pseudo code in listing 4.3 is not without issues. If there are only two modes (read and write), upgrading to a read from a write lock has a high probability of deadlocking. Consider two threads entering the do work method and assume that the monitor for the current object is unlocked. Both of these methods will successfully acquire the read lock on the current object and proceed to check the x value. Now assume that the value of the x field happens to be five. In this case both threads will try to upgrade their lock to a write lock. Unfortunately, the upgrade to a write lock requires that all readers release the read lock. This is not possible since both threads acquired read locks and wait for the other thread to release the read lock. Therefore the threads will deadlock and never finish execution.

The solution to this problem is to introduce another state for the lock called upgradable read mode. The upgradable read mode behaves just like the regular read mode when it comes to permissions. The major difference is, that threads which have acquired a regular read mode are not allowed to upgrade the lock to write mode. The threads which may upgrade to write mode are only threads which hold an upgradable read lock. Additionally the following rules are set on the locks.

In any given moment, there may only ever be one thread holding the upgradable read lock, but there may be arbitrary many threads holding a regular lock when a thread is holding an upgradable read lock. This is now sound as there will always be only one thread which will try to upgrade the lock mode to writable mode. Additionally, a thread holding an upgradable read mode lock, may downgrade to a regular read mode. This operation is non-reversible as a thread may not upgrade from a read lock to an upgradable read lock. Consider the pseudo code in listing 4.4.

```
1   method doWork()
2   {
3     acquire.upgradable this
4
5      if(this.x > 0){
6        upgrade this
7        this.x = 0;
8        downgrade this
9      }
10    downgrade this
11
12    /* Do some work here which requires *
13     * only read access to the field of x */
14
15    release this
16  }
```

Listing 4.4: Pseudo code showing read write and upgradable read locking in Chalice

	read lock	**upgradable lock**	**write lock**
upgrade	not allowed	write lock	not allowed
downgrade	not allowed	read lock	upgradable lock

Table 4.1: Resulting lock from calling *downgrade* or *upgrade*

Two new keywords *upgrade* and *downgrade* are introduced. These keywords are used to upgrade or downgrade a particular lock to a different level. Table 4.1 shows the resulting lock from using either the *upgrade* or *downgrade* statement. In general neither *upgrade* or *downgrade* may be used on a lock which is only a read lock, only *downgrade* may be used on the write lock and both statements can be used on an upgradable read lock.
The pseudo code in listing 4.4 is now thread safe. If two threads enter the *doWork()* method at once, both will try to capture the upgradable read mode. However, only one thread may acquire the upgradable mode at any time. Therefore only one thread will be able to proceed until the thread who has acquired the lock downgrades to a read lock. Then the other thread can acquire the upgradable read lock.

This approach does not suffer from any dead locks and maximizes the concurrency in the given application. Additionally, it does not require a lot of modification to the verifier, as upgrading just increments the permissions and downgrading decrements the permission.

Shadow Permissions

The last requirement to be introduced are shadow permissions. Chalice supports the concept of permissions, which can be used to read and write memory locations. These permissions will now be called *real* permissions. Shadow permissions are an extension of the concept, which allows for permissions, which can only be used to supplement regular permissions in order to write memory locations. In general shadow permissions may not be used to read any memory locations.

Shadow permissions follow similar rules to regular permissions. One would use the expression *acc(x, y, z)* to express shadow permissions. If *z* happens to be a hundred, then the shadow permissions become regular permissions. The *y* value may never be equal to a hundred, as this would create permissions with which anyone can write to a memory location, but no one could actually read the location.

As regular access permissions can be split into permissions fractions so can shadow permissions. If at a point in the program a thread holds five percent regular access permissions to a particular field, then these can be split in a hundred pieces of one percent shadow permissions. This would mean that a thread may gain access to one percent of 5 percent of access to a given field. The verifier when actually checking for permissions to write to a given location will now have to check the shadow permissions and add these to the real permissions. Additionally, shadow permissions can also be summed up to form the original real permissions. Consider the example in listing 4.5.

```
1  class Simple{
2    var x : int;
3    invariant acc(x, 95) && x > 0;
4
5    method setX(y : int)
6      requires rd(mu) && maxlock << this
7      requires rd(x, 5) //equivalent to acc(x, 5, epsilon)
8      ensures rd(x, 5)
9      ensures rd(mu)
10   {
11     acquire this;
12     //acc(x, 95) + rd(x,5) == 100% write permission
13     x := y;
14     release this;
15   }
16 }
```

Listing 4.5: Pseudo code showing how shadow permissions work

The monitor invariant only contains ninety-five percent access to the x field which means that acquiring the lock on the object alone is not enough to write to the field. Therefore the *setX* method additionally requires shadow permissions (of five percent real permissions) to be able to write to the memory location.

At a first glance, the shadow permissions do not seem to bring any benefit to Chalice. One could argue that the code provided in listing 4.6 is equivalent to the one provided in listing 4.5. As a matter of fact, the code in the method body behaves in exactly the same way.

```
1  class Simple{
2    var x : int;
3    invariant acc(x, 95) && x > 0;
4
5    method setX(y : int)
6      requires rd(mu) && maxlock << this
7      requires acc(x, 5)
8      ensures acc(x, 5)
9      ensures rd(mu)
10   {
11     acquire this;
12     x := y;
13     release this;
14   }
15 }
```

Listing 4.6: Pseudo code without shadow permissions

The major difference lies in the how many threads can call that particular method. If the contract used real permissions (listing 4.6) then at any given time, only one thread may have enough access

permissions to call that method, as permissions in Chalice cannot be forged. On the other hand, if the contract uses shadow permissions (listing 4.5) there may be an arbitrary number of threads which have access to the same object instance and may have the correct shadow permissions to actually call this method. Without the use of the shadow permissions, only one thread would be able to call the *setX()* method.

This approach is free from any race conditions as pure shadow permissions may not be used to read values from memory locations and only one thread may hold the remainder of the permissions which are required to write into that memory location.

4.2.2 Solution

This section describes a proposal for a solution for verifying the Composite pattern. Unfortunately, due to time constraints it was not possible to implement the solution and required prerequisites into Chalice. What follows is a description on how one would go about implementing the Composite pattern so that it can be verified in Chalice assuming all the requirements are met.

The solution adds to every composite object a field, called structure, which points to the owner. The owner can then be locked to change the particular mode of operation, which can be either read or write. In read mode, no objects in the structure may change and therefore all the reads will be consistent. In write mode threads are allowed to write to the composite objects. The remainder of the report will extend on the code provided explained in chapter 3.

Composite simplification

The Composite pattern had to be slightly modified in order to for the implementation to work. Section 4.1 gave ideas for representing structures for objects and the issues which accompany that approach. The biggest issue which could not be solved was keeping the lock order, e.g. a thread would have to set the desired mode to a particular structure before accessing the object itself.

To solve that particular issue, it was decided that the structure field is immutable as long as multiple threads may have access to the field. This is very similar in which the *mu* field works. Note, that this does not imply that the composite structure itself is immutable. Consider the fragment of code shown by listing 4.7.

```
1  class Composite{
2
3    var parent : Composite;
4    var left   : Composite;
5    var right  : Composite;
6
7    var total  : int;
8    ghost var index : int;
9
```

```
10    var structure : Structure;
11
12    invariant valid && correctTotal()
13
14    predicate valid
15    {
16      /*...*/
17      && acc(this.structure, 50) && rd(this.structure.mu) && this >> structure
18      && (left != null ==> acc(left.structure, 1) && structure == left.structure)
19      && (right != null ==> acc(right.structure, 1) && structure == right.structure)
20    }
21
22    /*...*/
23  }
```

Listing 4.7: Extension of Composite code to hold the invariant for structures

This invariant does restrict the concurrency slightly. In that setup it is not possible to add composite objects which already contain children, as that would break the invariant for the current object. Consider the fragment of code present in listing 4.8

```
1   method addLeft(newLeft : Composite)
2     /*...*/
3     requires acc(newLeft.structure, 100)
4     requires newLeft >> structure
5     requires !newLeft.hasLeft()
6     requires !newLeft.hasRight()
7
8     ensures acc(newLeft.structure, 49)
9   {
10    unfold valid
11    unfold newLeft.valid
12    left := newLeft;
13    left .parent := this;
14    left .structure := this.structure
15    left .index := 1
16    fold newLeft.valid
17    fold valid
18    release newLeft;
19    call addTotal(unfolding valid in newLeft.total)
20  }
```

Listing 4.8: Extension of Composite code showing how to add a new composite object to the structure

The code fragment shows the new additions to the old method contract. Most importantly, it is important to note that the method requires full access to the structure memory location of the object which is to be added. Additionally, the thread calling this method will then only be left with forty nine percent access to the structure field. This is because the monitor invariant itself holds fifty percent and the parent object of *newLeft* requires also at least read access to that field

to make sure that all objects which are linked together are in the same structure.

Now the thread which actually calls methods on the object, knows the structure field of that a particular object. Intuitively, there seems to be now another problem in actually sharing the access permissions to that particular field. In fact however, there are no additional restrictions placed on the code, since any thread wanting to lock any object would first have to acquire the mu field of said object. This is also the time when the thread can acquire access to the *structure* object.

Finally, that modification still does not guarantee absence of changes in read mode. The *addLeft* method has no precondition which would actually the code to first acquire a write mode on the current object.

Read/Write mode for the Structure

Having set up the code in such a way that would allow locking of the structure object before locking the actual object, it is possible to introduce multiple modes for the structure. The main modes which are interesting are read and write modes and how those could potentially be implemented in Chalice.

Firstly, let's remove some permissions from the composite object, so that locking the composite object is not enough to be able to write to any of the fields. It is enough to subtract 5 percent of access permissions from every field, which is in the same object, from the invariant. These permissions will now be stored inside the structure object. Therefore it is impossible to write to any of the fields of any composite object when the object is in a structure.

The next step is to set up the composite structure in such a way that setting the mode of the structure will return permissions to the caller such that having the mode set to read mode will result in fractions of real permissions. If the caller requested the write mode, then the resulting permissions will be shadow permissions.

Listing 4.9 shows pseudo code on how that functionality could be implemented in chalice.

```
1   class Permissions{
2     var structure : Structure;
3     var mode : int;
4     var users : int;
5
6     invariant acc(mode, 100);
7     invariant acc(structure, 50) && structure != null
8     invariant acc(structure.members, 100)
9     invairant acc(structure.mode, 10) && structure.mode == mode
10    invariant acc(users, 100)
11
12    invairant mode == 1 ==> acc(members[*].*, 5 − users * epsilon);
```

```
13    invairant  mode == 2 ==> acc(members[∗].∗, 5, 100 − users∗epsilon);
14  }
15
16  class Structure{
17
18    var members : seq<Composite>;
19    var mode : int;
20    var permissions : Permissions;
21
22    method setMode(x : int)
23      requires rd(mu)
24      requires maxlock << this.mu
25      requires x == 1 || x == 2
26      //where x == 1 ==> read mode
27      //  x == 2 ==> write mode
28
29      ensures maxlock == this.mu
30
31      //read permission to all member fields
32      ensures x == 1 ==> rd(members[∗].∗)
33
34      // shadow permissions to all member fields
35      ensures x == 2 ==> rd(members[∗].∗, 5)
36    {
37      acquire this.upgradable
38
39      if(mode != x){
40        upgrade this
41        acquire permissions
42        mode := x
43        permissions.mode := x;
44        release permissions
45        downgrade this
46      }
47
48      downgrade this
49
50      acquire permissions
51      permissions.users := permissions.users + 1
52      release permissions
53    }
54
55    method releaseMode()
56      requires rd(mu)
57      requires holds(this)
58      requires maxlock == this.mu
59      requires mode == 1 ==> rd(members[∗].∗)
60      requires mode == 2 ==> rd(members[∗].∗, 5)
61      ensures malock << this.mu
62    {
63      acquire permissions
64      permissions.users := permissions.users − 1
```

```
65          release permissions
66          release this
67      }
68  }
```

Listing 4.9: Pseduo code for the structure

From the client perspective, there are not too many changes. The *addLeft()* method would additionally require at least shadow permissions to all the fields the method writes to. The client can only gain those permissions by setting the correct mode on the structure. On the other hand, if the client sets the mode to read mode, there is no need for the client to do any locking of the composite objects anymore. The client will be able to read any value from any object in the structure.

Chapter 5

Additional Work

This chapter of the report describes the additional work which was done during the project, which did not contribute directly to the main goal. The first section 5.1 briefly describes the Chalice plugin which was developed for Eclipse. Section 5.2 quickly described the bugs which were found while working with Chalice and finally finishes with section 5.3 describing some changes which could make Chalice even more interesting and useful.

5.1 Eclipse Plugin

This section describes the Eclipse plugin which adds support for Chalice into the Eclipse integrated development environment. Eclipse is a development environment almost fully written in Java and initially developed for Java. Eclipse was developed to be plugin based and soon new plugin arrived adding support for other different languages such as C, C++, Python and many more. Eclipse is released under the terms of the Eclipse public License, which makes the software open source and therefore easier to develop plugins for.

Figure 5.1 shows a screen shot of Eclipse running the Chalice plugin with an open Chalice program file. Note, that the Eclipse window was resized so that it could fit the page. In common usage there is a lot more space provided in the frames. The following features are supported by the plugin.

- Syntax highlighting of keywords
- Syntax Parsing on the fly when typing. This allows for displaying syntax errors while the user types
- Outline which shows the building blogs of the program, including classes, predicates, methods, functions, variables, etc.
- Running the Chalice compiler from within the IDE
- Highlighting errors found by the Chalice compiler in the text

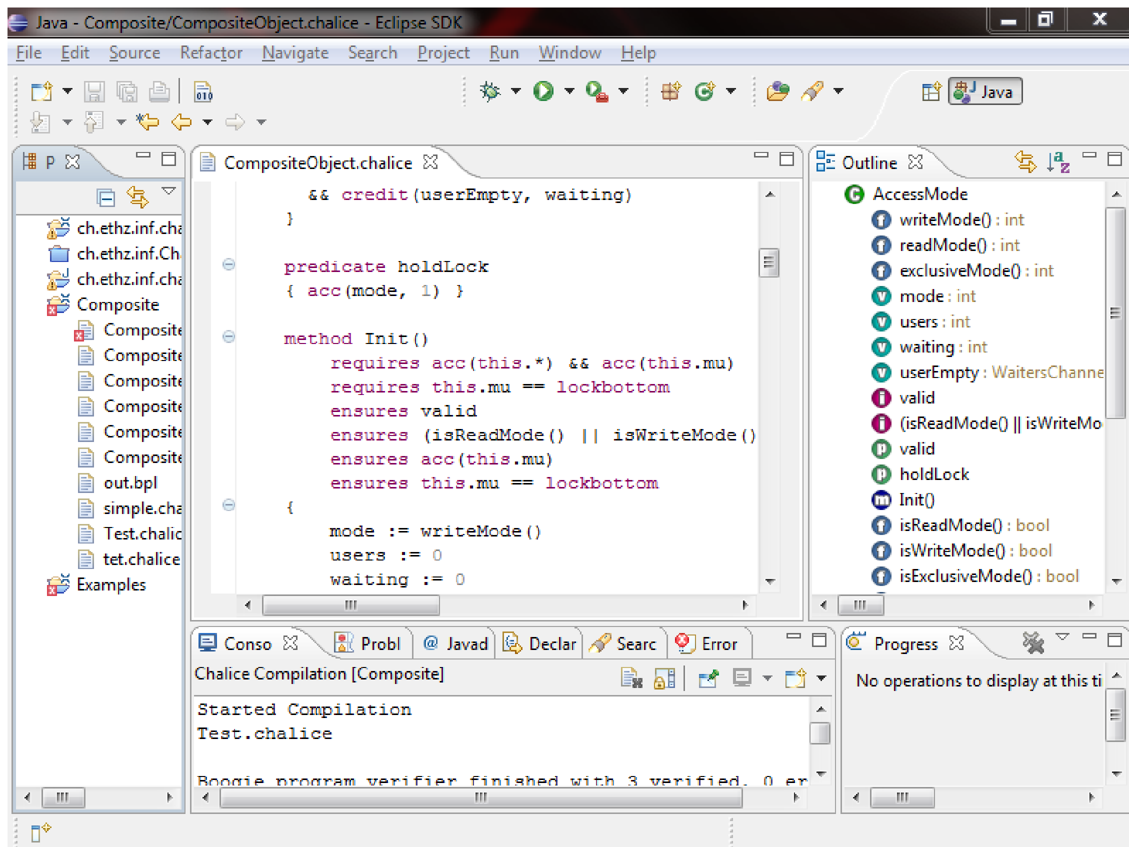

Figure 5.1: A screen shot of Eclipse running the Chalice plugin

- Hyperlinks from the compiler output to the Chalice program file
- Very basic auto completion of code

All these features have been very helpful while working on the project. The two most useful features were certainly the syntax error highlighting while typing and the invoking of the Chalice compiler from within the IDE. The syntax error highlighting was useful because it would give immediate feedback of syntactically incorrect Chalice program. This was most helpful at the beginning where one would easily revert back to using Java styles syntax.

The invocation of the Chalice compiler gave the Eclipse plugin a very important functionality. The output of the Chalice compiler was post processed by the plugin, allowing hyperlinks from the output to the program file. Therefore, the programmer does not need to manually count the line numbers which Chalice has objections to.

5.2 Finding Bugs

This section of the report describes bugs which have been found during the project. The bugs which were already discussed in the report will not be listed here.

5.2.1 Generated Chalice Code uses default package

Chalice was developed using Scala, a general purpose language compiling to Java bytecode. Therefore any Java Virtual Machine (JVM) can run the Chalice compiler if the Scala libraries are present. The problem with Chalice was that the code produced belonged to the default package and therefore disabled any properly designed Java program to call the compiler from within the same JVM. This bug was not a major issue, but has been fixed in the latest version of Chalice.

5.2.2 Nullpointer exception at function post condition

It would seem that at one point Chalice would not ignore post conditions for function and would try to do some computation with the post condition of a function. Consider the following code.

```
1
2  class Test{
3    var tests : seq<Test>;
4    var total : int;
5
6    invariant acc(tests, 100);
7    invariant acc(total, 50);
8
9    function at(loc : int) : Test
10     requires acc(tests);
11     requires loc >= 0 && loc < size();
12   {
13     tests[loc]
```

```
14    }
15
16
17    function size() : int
18      requires acc(tests);
19      ensures result >= 0;
20       // PROBLEM
21      ensures result == |tests|; //nullpointer
22    {
23      | tests |
24    }
25
26    predicate pre
27    { acc(total, 50) }
28  }
```

The compiler did not handle the *ensures result == |tests|* very well and would crash with a Java NullPointer Exception. This bug has also been fixed after it was reported.

5.2.3 Lockchange on return variables

Chalice allows to use the *lockchange* expression on variables which are return variables of methods. This has been also done in the Chalice examples. The issue here was that the examples never actually called the methods which would define the *lockchange* expression on return values as these methods were designed to be main methods, methods only called at the program entry. The issue only appears if the method is actually called, which will then cause Chalice to fail. The following code will trigger the issue.

```
1   class Bug{
2
3     method Main()
4     {
5       var a : Bug;
6       call a:= m();
7     }
8
9     method m() returns (a : Bug)
10
11      lockchange a
12    {
13      a := new Bug;
14      share a;
15      acquire a;
16    }
17  }
```

This bug was not yet fixed.

5.3 Interesting Chalice features

During the project, there were found some quite interesting features which could make Chalice even more useful than it is. This section briefly describes those features.

5.3.1 Predicates and *mu* fields

A very important part of programming languages is information hiding. In Chalice predicates are used to hide information about the implementation so that an external class does not need to consider what the invariants actually mean, but just needs to know that they need to be somehow ensured.

On the other hand, there exists the *mu* field which is actually not implementation dependent and it always has the same meaning. Unfortunately, Chalice right now forces the code to hide any access permission to the *mu* inside the predicate if the predicate contains a reference to the *mu* *field. This is very unfortunate because the external client code may have good reasons to manage all the access permissions to the* mu *field of a particular object. This usually happens when a thread wants to reorder a given object so that it has a different* mu *value. To do this operation, the thread needs to acquire all access permissions to the field, which may be hard to do, since the information about the* mu *field might be hidden away in predicates.*

Chalice also faces another issue. If a predicate requires permissions to the mu *field and the invariant of an object consists of that predicate, it becomes impossible to share the object. This is due to the fact that any predicate, which exists in the monitor invariant, must be folded before the object can be shared. Once the predicate is folded, the permissions to the* mu *field are consumed, and the current program point does not have a hundred percent access to the* mu *field, and therefore is unable to share the object.*

This is very unfortunate, as at that point in time only one thread has access to that variable. Moreover, if the program is rewritten not to use predicates, but invariants which show all the information, the program verifies without a problem. Therefore, it seems that it would actually make sense to distinguish between public parts of a predicate and private parts of a predicate.

5.3.2 Destroying access permissions

Chalice does not currently support constructors. It would seem to be of benefit to add constructors to Chalice, which could destroy fraction of access permissions on purpose. The real benefit from destroying permissions comes from the fact that if no thread can obtain full permission to a given field, this field will never change and therefore becomes a constant value. Therefore, threads would not require any permission at all to read memory locations which they know are constant.

Chapter 6

Conclusion

The verification of the Composite pattern in a concurrent setting turned out to be quite a challenging task. Most of the time spent on the project was used to investigate different ideas which would help in verifying the Composite pattern. It turned out that most ideas either had issues which would not be easily solved, concepts would be too complex to implement and give too little benefit to Chalice and some other ideas would simply not work at all.

There was one Chalice Composite pattern implementation done, which was successfully verified by Chalice. That implementation supports only writing to Composite objects and keeping the monitor invariant. Unfortunately, due to time constraints, it was not possible to implement the a version of the Composite pattern which would support writing to the objects and consistent reading from the objects. It was, however, possible to give a detailed outline on how one would go abut about implementing that Composite pattern instance so that it can be verified by Chalice.

Early on the project some time was spent to build the Eclipse plugin. This turned out to be a major time saver when working on the implementations of the Composite pattern. Thanks to the ability to quickly show syntax errors and the ability of invoking the Chalice compiler from within the editor saved a lot of time from tedious switching of windows and executing command line commands.

Future work certainly includes finishing the implementation of the Composite pattern which supports reading consistent values and not just writing. To implement the proposed approach, it would also be necessary to modify Chalice to support the prerequisites of said approach. In general, it seems that these prerequisites are certainly implementable in Chalice.

Finally, summing up the project, it seems that it has been successful, although there is no concrete implementation of the Composite pattern which would support all the required functionality, there is a detailed description on how one would go about implementing the Composite pattern which would be verifiable by Chalice.

Appendix A

A simple implementation of the Composite pattern

```
1  class Composite{
2
3    var parent  : Composite;
4    var left    : Composite;
5    var right   : Composite;
6
7    var total  : int;
8    ghost var index  : int;
9
10   invariant valid && correctTotal()
11
12   /*
13     PUBLIC
14   */
15
16   predicate valid
17   {
18     acc(parent, 50) && acc(right, 50) && acc(index, 50) && acc(left,50) && acc(total, 50)
19
20     && (index == 0 || index == 1 || index == 2)
21     && (index == 0 ==> parent == null)
22     && ((index == 1 || index == 2) ==> parent != null )
23
24     && (left != null ==> acc(left.total, 50) && acc(left.parent, 50) && left.parent == this
           && acc(left.index, 50) && left.index == 1)
25     && (right != null ==> acc(right.total, 50) && acc(right.parent, 50) && right.parent ==
          this && acc(right.index, 50) && right.index == 2)
26     && (parent != null ==> ite(index == 1, acc(parent.left, 50), acc(parent.right, 50)))
27     && (parent != null ==> acc(mu,1) && acc(parent.mu,1) && this << parent)
28
29     && (index == 1 ==> parent.left == this)
30     && (index == 2 ==> parent.right == this)
31
```

```
32      && (left == null ==> acc(left, 50))
33      && (right == null ==> acc(right,50))
34      && (parent == null ==> acc(total, 50) && acc(parent, 50) && acc(index, 50))
35    }
36
37    method Init()
38      requires acc(parent) && acc(left) && acc(right) && acc(total) && acc(index);
39      requires acc(mu);
40      requires mu == lockbottom
41
42      ensures valid
43      ensures acc(mu, 100)
44      ensures mu == lockbottom
45      ensures correctTotal()
46    {
47      parent := null;
48      left := null;
49      right := null;
50      total := 1;
51      index := 0;
52      fold valid
53    }
54
55    method addLeft(newLeft : Composite)
56      requires newLeft != null
57
58      requires holds(this)
59      requires holds(newLeft)
60
61      requires valid
62      requires newLeft.valid
63
64      requires !hasLeft()
65      requires newLeft.addable()
66
67      requires correctTotal()
68      requires newLeft.correctTotal()
69
70      requires acc(mu,1)
71      requires acc(newLeft.mu,1)
72      requires newLeft << this
73      requires maxlock == this.mu
74
75      lockchange this
76      lockchange newLeft
77    {
78      unfold valid
79      unfold newLeft.valid
80      left := newLeft;
81      left.parent := this;
82      left.index := 1
83      fold newLeft.valid
```

```
84      fold valid
85      release newLeft;
86      call addTotal(unfolding valid in newLeft.total)
87    }
88
89    function hasLeft() : bool
90      requires valid
91    {
92      unfolding valid in left != null
93    }
94
95    function addable() : bool
96      requires valid
97    {
98      unfolding valid in parent == null
99    }
100
101
102   function correctTotal() : bool
103     requires valid
104   {
105     unfolding valid in (total == (1 + ite(left == null, 0, left.total) + ite(right == null,
          0, right.total)))
106   }
107
108   function getLeft() : Composite
109     requires valid
110   {
111     unfolding valid in left
112   }
113
114   /*
115     PRIVATE
116   */
117
118   function correctTotalAmount() : int
119     requires valid
120   {
121     unfolding valid in (1 + ite(left == null, 0, left.total) + ite(right == null, 0, right.
          total))
122   }
123
124   method addTotal(amount : int)
125     requires holds(this)
126     requires valid
127
128     //requires unfolding valid in maxlock == this.mu
129     requires unfolding valid in (parent != null ==> maxlock << parent.mu)
130     requires (unfolding valid in total) == correctTotalAmount() − amount
131     lockchange this
132   {
133     var current : Composite := this
```

```
134        var parent : Composite;
135        while(current != null)
136          invariant current != null ==> holds(current)
137          invariant current != null ==> current.valid
138          //invariant current != null ==> unfolding current.valid in maxlock == current.mu
139          invariant current != null ==> unfolding current.valid in (current.parent != null ==>
                 maxlock << current.parent.mu)
140          invariant current != null ==> (unfolding current.valid in current. total ) == current.
                 correctTotalAmount() − amount
141          //lockchange current
142        {
143          unfold current. valid
144          parent := current.parent;
145          if(parent != null){
146            acquire parent
147            unfold parent. valid
148          }
149          current. total := current. total + amount;
150          fold current. valid
151          release current
152
153          current := parent
154          if(parent != null){
155            fold current. valid
156          }
157        }
158      }
159    }
160
161    class Program{
162
163      method Main() returns (root : Composite, l1 : Composite, l2 : Composite, l3 : Composite)
164      {
165        root := new Composite;
166        l1 := new Composite;
167        l2 := new Composite;
168        l3 := new Composite;
169
170        call l1 . Init ()
171        call l2 . Init ()
172        call l3 . Init ()
173        call root . Init ()
174
175        share l1
176        share l2 above l1
177        share l3 above l2
178        share root above l3
179
180        fork addLeft(root , l3 )
181        fork addLeft(l2 , l1 )
182        fork addLeft(l3 , l2 )
183      }
```

```
184
185     method addLeft(parent : Composite, child : Composite)
186       requires parent != null && child != null
187       requires acc(parent.mu,1)
188       requires acc(child.mu,1)
189
190       requires maxlock << child.mu && child.mu << parent.mu
191       requires maxlock << parent.mu
192
193       lockchange parent
194       lockchange child
195     {
196       acquire child
197       acquire parent
198       if(!parent.hasLeft() && child.addable()){
199         call parent.addLeft(child)
200       }else{
201       release parent
202       release child
203       }
204     }
205   }
```

Bibliography

[1] *Gary Leavens, K. Leino, and Peter Müller. Specification and verification challenges for sequential object-oriented programs.* Formal Aspects of Computing*, 19:159–189, 2007.*

[2] *K. Leino, Peter Müller, and Jan Smans. Verification of concurrent programs with chalice. In Alessandro Aldini, Gilles Barthe, and Roberto Gorrieri, editors,* Foundations of Security Analysis and Design V*, volume 5705 of* Lecture Notes in Computer Science*, pages 195–222. Springer Berlin / Heidelberg, 2009.*

[3] *Steam. Cpu count in modern machines, July 2010.* `http://store.steampowered.com/hwsurvey`.

[4] *Alexander Summers and Sophia Drossopoulou. Considerate reasoning and the composite design pattern. In Gilles Barthe and Manuel Hermenegildo, editors,* Verification, Model Checking, and Abstract Interpretation*, volume 5944 of* Lecture Notes in Computer Science*, pages 328–344. Springer Berlin / Heidelberg, 2010.*