Verification of Concurrent Programs with Chalice

K. Rustan M. Leino[0], Peter Müller[1], and Jan Smans[2]

[0] Microsoft Research, Redmond, WA, USA, leino@microsoft.com
[1] ETH Zurich, Switzerland, peter.mueller@inf.ethz.ch
[2] KU Leuven, Belgium, jan.smans@cs.kuleuven.be

Abstract. A program verifier is a tool that allows developers to prove that their code satisfies its specification for every possible input and every thread schedule. These lecture notes describe a verifier for concurrent programs called Chalice. Chalice's verification methodology centers around permissions and permission transfer. In particular, a memory location may be accessed by a thread only if that thread has permission to do so. Proper use of permissions allows Chalice to deduce upper bounds on the set of locations modifiable by a method and guarantees the absence of data races for concurrent programs. The lecture notes informally explain how Chalice works through various examples.

0 Introduction

Writing correct sequential programs is *hard*. The correctness of a program typically relies on many implicit assumptions. For example, a developer may assume that a certain variable lies within the bounds of an array, that a parameter is non-null, that a method will not modify the contents of a certain memory location, or that operations on a given data structure follow some particular protocol. However, it is difficult for developers to see whether their assumptions are correct and, as a consequence, it is difficult to write correct programs.

Writing correct concurrent programs is *even harder* because of data races, deadlocks, and potential interference among threads. That is, developers not only need to worry about the assumptions needed for reasoning about sequential programs, but additionally have to make assumptions about the effects of other threads. For example, to avoid data races, programmers need to ensure, at each field access, that other threads will not access the same location concurrently. Similarly, when a lock is acquired, one must determine what memory locations are protected by that lock and whether the acquisition may lead to deadlock.

In this paper, we describe Chalice, a language and program verifier that detects bugs in concurrent programs. More specifically, developers can make the implicit assumptions about their code explicit via annotations, so-called *contracts*. For example, annotations can indicate that a parameter must not be null, that a memory location can safely be accessed by a thread, or that a memory location is protected by a certain lock. The Chalice program verifier analyzes annotated programs and checks that the given annotations are never violated.

Let us look at the small sequential program of Fig. 0, to get a feeling for what the annotations look like and how Chalice works. The program consists of a class `Math` with

```
class Math {
  method ISqrt(n: int) returns (res: int)
    requires 0 <= n;
    ensures res * res <= n && n < (res+1) * (res+1);
  {
    res := 0;
    while ((res + 1) * (res + 1) <= n)
      invariant res * res <= n;
    {
      res := res + 1;
    }
  }
}
```

Fig. 0. A small sequential program where assumptions have been made explicit via annotations.

a single method `ISqrt` whose goal is to compute and return the positive integer square root of its parameter `n`. `ISqrt`'s implementation achieves this goal by incrementing the output parameter `res` until the desired value is reached.

Fig. 0 contains three annotations that explicate assumptions about `ISqrt`.

First, `ISqrt` has a *precondition*, starting with the keyword **`requires`**, which indicates that `ISqrt`'s developer expects callers to supply a non-negative value for `n`. Chalice checks this assumption at each call site. For example:

```
call x := math.ISqrt(5); // ok
call x := math.ISqrt(-5); // error: unsatisfied precondition 0 <= n
```

Second, `ISqrt` has a *postcondition*, starting with the keyword **`ensures`**, which specifies that the output parameter `res` will contain the integer square root of `n` on exit of `ISqrt`. Chalice checks the method's implementation to satisfy this assumption whenever the implementation returns to its caller.

Third, `ISqrt`'s body has a *loop invariant*, starting with the keyword `invariant`, which declares that every time execution reaches the head of the loop, the square of `res` is at most `n`. In general, a loop invariant must hold on entry to the loop and must be preserved by the loop's body. If this is not the case, then Chalice reports an error.

The program of Fig. 0 is sequential and does not use the heap. Therefore, we did not have to worry about data races, deadlocks, or what part of the heap the method modifies. In the remainder of this paper, we describe how Chalice verifies assumptions about complex concurrent programs that use shared mutable state by relying on a system of permissions and permission transfer. Section 1 describes the heart of the Chalice annotation methodology, namely permissions and permission transfer. Section 2 then shows how permissions can be used to determine what locations a method can modify. We discuss fork-join parallelism in Section 3, extend this approach to monitors in Section 4, and show how to avoid deadlocks in Section 5. Data abstraction is handled in Section 6. We show how Chalice verifies a classic example from Owicki and Gries [34]

in Section 7 and tackle rely-guarantee reasoning in Section 8. Finally, we conclude in Section 9.

This paper informally describes Chalice through various examples. The goal here is to explain common usage of the annotation style, not to provide a reference manual for the language and program verifier. For more information on how Chalice works internally and on other features not discussed here, we refer the reader to [30].

1 Permissions

Verification in Chalice centers around permissions. Permissions can be held by an *activation record*, that is, a particular invocation of a method. A memory location may be read or written by an activation record only if that activation record has permission to do so. We denote permission to access field `f` of object `o` by **`acc`**`(o.f)`.

Let's look at an illustrative example. Consider the program of Fig. 1. At the start of this program, the stack contains a single activation record for `Main` that holds no permissions. When a new object is created, the creating activation record gains permission to access the fields of the new object. For example, the first statement of `Main` gives the activation record permissions to access `c.x` and `c.y`. Because of those permissions, the activation record can safely update `c.x` and read `c.y` on the next line.

The next statement in `Main` is a method call: **`call`** `c.Inc()`. Execution of a method call starts by pushing a new activation record onto the call stack. Which permissions does the new activation record have initially? The answer to this question can be determined by looking at `Inc`'s precondition, which indicates (0) that the caller must have permission to access **`this`**`.x` and (1) that this permission transfers from the caller's activation record to the callee's. In a similar fashion, the postcondition indicates which permissions transfer from the callee back to the caller on exit of `Inc`. In our example, this annotation returns to callers the permission to access **`this`**`.x`. Thus, `c.Inc()` temporarily borrows the permission to access `c.x` from `Main` such that it can read and update this memory location. In other words, the main method loses access to `c.x` for the duration of the call. It does, however, retain permission to access `c.y`. The assignment to `c.x` after the call is safe because `Main`'s activation record once again has access to both `c.x` and `c.y`.

The last statement of `Main` is again a method call. This call consumes the permissions to `c.x` and `c.y`: `Dispose`'s precondition causes both permissions to be transferred to the callee, but the postcondition does not return any permissions to the caller.[0] As a consequence, the main method is not allowed to access `c`'s fields after calling `Dispose`. If `Main` attempted to update `c.x` after calling `Dispose`, Chalice would report the following error: `The location c.x might not be writable.`

The program of Fig. 1 successfully verifies. That is, Chalice outputs the following message after analyzing Fig. 1. (Boogie is the name of the program verifier Chalice is built on. The seven verified methods correspond to the three methods of class `C` plus four methods that are generated internally to check the well-formedness of `C`'s contracts.)

[0] These permissions are lost forever. Chalice offers a `-checkLeaks` mode that reports errors when permissions are permanently lost, but we assume the default mode in this paper.

```
class C {
  var x: int; var y: int;

  method Main() {
    var c := new C;
    c.x := 7; var tmp := c.y;
    call c.Inc();
    c.x := c.y + 10;
    call c.Dispose();
  }

  method Inc()
    requires acc(x);
    ensures acc(x);
  { x := x + 1; }

  method Dispose()
    requires acc(x) && acc(y);
    ensures true;
  { }
}
```

Fig. 1. A program that illustrates permissions and permission transfer.

```
Boogie program verifier finished with 7 verified, 0 errors.
```

However, what happens if we remove the precondition from `Inc`? Removing this annotation causes the verification of `Inc`'s body to fail, as the activation record then does not have permission to access `x`. What happens if we leave the precondition in place, but remove `Inc`'s postcondition? Removing this annotation causes the verification of `Main`'s body to fail, as `Main` then has insufficient permissions to perform the assignment `c.x := c.y + 10`.

Permissions can neither be duplicated nor forged. Because permissions cannot be duplicated, the stack can never contain permission to access a location more than once.[1] This means that a method with precondition **`requires acc`**`(o.f) && acc(o.f)` can never be called, since no caller is able to provide permission for accessing `o.f` twice.

It is important to understand that permissions are purely conceptual. They are needed only for verification, but have no effect during programs execution. Therefore, permissions can be mentioned only inside annotations but not inside method bodies.

As another example, consider the method `ISqrt` shown in Fig. 2. The major difference between this version and the one of Fig. 0 is the use of the heap. That is, instead of input and output parameters, Fig. 2 computes the integer square root of a field `n` and stores the result in `n` itself. A consequence of using a heap cell instead of a parameter is that the pre- and postconditions must require and return permission to access `n`, respectively. In addition, the loop invariant must demand access to `n` as well, because the body

[1] We relax this rule in Section 3.2 where permissions can be split into fractions.

```
class Math {
  var n: int;

  method ISqrt()
    requires acc(n) && 0 <= n;
    ensures acc(n) && n*n <= old(n) && old(n) < (n+1)*(n+1);
  {
    var N := n;
    n := 0;
    while ((n+1)*(n+1) <= N)
      invariant acc(n) && n*n <= N;
    {
      n := n + 1;
    }
  }
}
```

Fig. 2. A variant of the method `ISqrt` from Fig. 0 that uses a field instead of parameters.

of the loop is verified assuming only the loop condition and the loop invariant—in this regard, each loop iteration is treated as an activation record. If we removed the conjunct `acc(n)` from the loop invariant, then the body would not be allowed to assign to `n`.

As illustrated by the contract of class `Math`, annotations typically consist of *access assertions*, which specify the required or guaranteed access permissions, and *pure assertions*, which specify properties of variables. A pure assertion in an annotation may refer to a heap location `o.f` only if that annotation also contains an access assertion for `o.f`. Intuitively, one can think of the annotation as reading `o.f` and, therefore, it must have the permission to do so. Chalice rejects annotations that do not include the appropriate access assertions. For instance, if we omit `acc(n)` from `ISqrt`'s postcondition, Chalice reports the following error:

```
6.18: Location might not be readable.
```

The expression `old(n)` in the postcondition of `ISqrt` represents the value of the field `n` on entry to the method. In general, `old(e)` is the value that the expression `e` had in the method pre-state.

In summary, a heap location may be accessed by an activation record only if that activation record has permission to do so. An activation record can gain access to a memory location in one of three ways. First, the activation record can create a new object and thereby gain access to the fields of this new object. Second, a method can demand, via its **`requires`** annotation, that its caller provide certain permissions. The corresponding permissions are transferred from the caller on entry to the method. Third, an activation record can gain access to a memory location by calling a method that returns permission to access the memory location, as described in its postcondition.

In the next sections, we show why permissions are useful when reasoning about imperative programs. In particular, we demonstrate that one can derive what locations

```
class RockBand {
  var memberCount: int;
  invariant acc(memberCount) && 0 <= memberCount;

  method Init()
    requires acc(memberCount);
    ensures acc(memberCount) && memberCount == 0;
  { memberCount := 0; }

  method AddMembers(howMany: int)
    requires acc(memberCount) && 0 <= howMany;
    ensures acc(memberCount);
    ensures memberCount == old(memberCount) + howMany;
  { memberCount := memberCount + howMany; }
}
```

Fig. 3. The class `RockBand`. We will explain the meaning of the invariant in Section 4.

a callee can modify from the required permissions (Section 2) and that proper use of permissions prevents data races (Sections 3 and 4).

2 Framing

An important problem in the verification of imperative programs is the *frame problem*: a method contract must not only describe how it affects the program state (typically via pre- and postconditions), but must also make manifest what part of the heap it will definitely not modify. For example, before the call to `Inc` in the `Main` method of Fig. 1, `c.y` and `tmp` hold the same value. How do we know, only by looking at `Inc`'s contract, that `c.Inc()` does not assign a new value to `c.y`?

How can permissions help in solving the frame problem? The answer is that we can deduce an upper bound on the set of locations that can be modified by a method from the permissions that it requires. More specifically, if a callee does not demand access to a memory location to which the caller has access, then that callee cannot modify that location—it did not have the permission initially and it is impossible to forge permissions. For example, the call `c.Inc()` cannot modify `c.y`, since `Inc` requests access only to `c.x`. Therefore, we can show that `c.y` still equals `tmp` after the call.

Let's look at another example. Consider the class `RockBand` from Fig. 3. `RockBand` provides methods to initialize a band and to add band members. `Init` requires access to `memberCount` and returns this permission to its caller. It also ensures that the value of field `memberCount` is `0`. Similarly, `AddMembers` requires access to `memberCount` and additionally requires that a non-negative number is passed for the parameter `howMany`. The `Main` method of Fig. 4 shows how permissions solve the frame problem. At program location `A`, the postcondition of `AddMembers` holds: `Main` has regained access to `acdc.memberCount` and the field holds the value `5`. Creating a new object does

```
class Program {
  method Main() {
    var acdc := new RockBand;
    call acdc.Init();
    call acdc.AddMembers(5); // A

    var noDoubt := new RockBand;
    call noDoubt.Init();
    call noDoubt.AddMembers(4);

    assert acdc.memberCount == 5;
  }
}
```

Fig. 4. A client for `RockBand` that illustrates how permissions solve the frame problem.

not affect the values of existing locations; hence, after creating `noDoubt`, the location `acdc.memberCount` still holds `5`. The next two statements are method invocations whose preconditions do not request permission to access `acdc.memberCount`. Therefore, the caller never relinquishes its permissions to `acdc.memberCount`, from which one can conclude that `acdc.memberCount` is not modified by the two calls; hence, `acdc.memberCount` does equal `5` when the **`assert`** statement is reached.

3 Threads

So far we have only considered sequential programs. In this section, we introduce threads, explain how permissions are transferred when threads are forked and joined, and introduce read permissions, which allow several threads to read a location concurrently.

3.0 Creating and Joining Threads

A concurrent program has multiple threads of control. In Chalice, new threads are introduced using explicit **`fork`** statements. A **`fork`** statement indicates the method to be used as the starting point of the new thread, and it passes parameters and permissions to this method just as is done for an ordinary method call. For example,

```
fork c.Inc();
```

creates a new thread that will execute the `Inc` method on object `c`.

Using the **`join`** statement, one thread can wait for another to complete. The statement takes as argument a *token*, which uniquely identifies a thread. The **`fork`** statement creates the token, and allows it to be recorded in a variable:

```
fork tk := c.Inc();
```

For convenience, if the left-hand side of this assignment is not already a local variable, then the **fork** statement implicitly introduces the given identifier as a new local variable of the appropriate type. A thread can be joined only once.[2]

The **join** statement arranges for the permissions entailed by the method's postcondition to be transferred to the joining thread. In fact, a **call** statement is nothing but a **fork** immediately followed by a **join**. That is, **call** `c.Inc();` will have the same effect as

```
fork tk := c.Inc(); join tk;
```

except possibly for performance differences. Any expression **old**(E) in the method's postcondition refers to the value of expression E at the time of the **fork**.

If the method has output parameters, these are returned at the time of the join. The syntax is

```
join x,y,z := tk;
```

where x, y, and z are local variables that receive the actual output parameters.

3.1 Transferring Permissions to Threads

Because the **fork** statement transfers permissions, the permissions entailed by the preconditions of methods forked concurrently cannot overlap. For example, the Inc method in Fig. 1 lists **acc**(x) in its precondition; therefore, it is not possible to fork two concurrent copies of `c.Inc()`:

```
fork c.Inc();
fork c.Inc();  // error: unsatisfied precondition acc(c.x)
```

Figure 5 shows an example program that uses two concurrent threads to compute the integer square root of two numbers. One thread uses the permission on `m0.n` and the other one `m1.n`, and thus, the two threads operate on disjoint data. An attempt to assign to `m0.n` just after forking `m0.ISqrt()` would result in a verification error, since the fork operation transfers the calling thread's permission to `m0.n` (as indicated by the precondition of `Math.ISqrt`). However, after joining that thread, which is done by supplying the **join** statement with the token `tk0` that resulted from the fork operation, the calling thread regains the permission to `m0.n`, as indicated by `ISqrt`'s postcondition.

3.2 Read Permissions

In the previous subsection, we showed how to start threads and pass them the permissions to disjoint sets of memory locations. Sometimes, it is desirable to let several threads access the same memory location. Such mutual access is harmless as long as no thread updates the memory location. For this reason, for every memory location

[2] Whether or not a token `tk` is joinable is recorded in the ghost variable `tk.joinable`, which can be mentioned in program expressions.

```
class ParallelMath {
  method TwoSqrts(x: int, y: int) returns (r: int, s: int)
    requires 0 <= x && 0 <= y;
    ensures r*r <= x && x < (r+1)*(r+1);
    ensures s*s <= y && y < (s+1)*(s+1);
  {
    var m0 := new Math { n := x };
    var m1 := new Math { n := y };
    fork tk0 := m0.ISqrt();
    fork tk1 := m1.ISqrt();
    join tk0;
    join tk1;
    r := m0.n;
    s := m1.n;
  }
}
```

Fig. 5. An example program for computing two square roots in parallel. The `Math.ISqrt` method is defined in Fig. 2.

`o.f`, Chalice distinguishes between full permission to `o.f`, written **acc**`(o.f)`, and read permission to `o.f`, written **rd**`(o.f)`.

We can think of permissions as percentages [6]. Full permission corresponds to 100% of a memory location's permission, whereas a read permission corresponds to an ε of the location's permission, for some infinitesimal ε (that is, some arbitrarily small, yet positive, value ε).

In order for an activation record to write a memory location `o.f`, it must hold 100% of the permission to `o.f`, whereas reading `o.f` can be done with any non-zero amount of permission to `o.f`. An activation record that holds 100% permission to `o.f` can transfer any number of ε's of that permission to other activation records and threads; doing so leaves the activation record with less than 100% permission, which means it can no longer write `o.f`. If the activation record manages to collect back all of the ε's given out, it will once again have 100% access and can then write `o.f`. Note that while one thread has a read (that is, non-zero) or write (that is, 100%) permission to a location `o.f`, no other thread can have write permission to `o.f`, which prevents data races. That is, the sum of the permissions to `o.f` across the activation records of all threads can never exceed 100%.

As an example, Fig. 6 shows two classes that implement a part of a video store, and in particular a part of the video store's way of charging for video rentals. A video rental indicates a customer and a movie, as well as the duration of the rental, counted in number of days. The method that computes the frequent rental points awarded for this rental requires read access to `customerId` and `movieId`, and the method that produces the invoice for the rental requires read access to `movieId` and `days`. The `Charge` method of class `VideoStore` can therefore compute frequent rental points and produce the invoice concurrently.

```
class VideoRental {
  var customerId: int;
  var movieId: int;
  var days: int;

  method FrequentRentalPoints() returns (points: int)
    requires rd(customerId) && rd(movieId);
    ensures rd(customerId) && rd(movieId);
  {
    // ...
  }

  method Invoice() returns (dollars: int)
    requires rd(movieId) && rd(days);
    ensures rd(movieId) && rd(days);
  {
    // ...
  }
}

class VideoStore {
  method Charge(vr: VideoRental)
    requires acc(vr.customerId) && acc(vr.movieId) && acc(vr.days);
  {
    fork tk0 := vr.FrequentRentalPoints();
    fork tk1 := vr.Invoice();
    var p: int; var d: int;
    join p := tk0;
    join d := tk1;
    // ...
  }
}
```

Fig. 6. An example that shows the use of **rd** permissions. The two threads created by the `Charge` method both have read access to `movieId`, but neither thread can write it.

The Charge method in Fig. 6 creates one thread for each of the two operations and then just waits for the two threads to finish. This looks symmetric and nice, but it uses more threads than needed. An alternative, which uses a total of only two threads instead of the calling thread plus two worker threads, is the following:

```
fork tk := vr.FrequentRentalPoints();
call d := vr.Invoice();
join p := tk;
```

The **rd** annotation, which specifies ε permissions, is often convenient when indicating read access. If a program gives read access to a location and later wants to regain write access, it must arrange to collect all the ε permissions that were given out. For example, after the two **fork** statements in Fig. 6, the calling thread has $100\% - \varepsilon - \varepsilon$ access to `vr.movieId` and each of the other threads holds an ε permission. Chalice allows these permissions to be split up in other ways, too, using the **acc**`(o.f, n)` annotation, which indicates `n`% of permissions. For example, instead of **rd**`(movieId)`, method `FrequentRentalPoints` could have said **acc**`(movieId,17)` and method `Invoice` could have said **acc**`(movieId,31)`, which would have left the calling thread with 52% after the two **fork** operations. It is sometimes convenient to split the permissions this way (as we shall soon see), but note that 99% permission and ε permission both provide just read access—the difference lies in the ease of writing specifications that give the program the opportunity to assemble back the entire 100% of the permissions. In light of this discussion, **rd**`(o.f)` can be viewed as a way to write **acc**`(o.f, ε)`.

4 Monitors

The machinery introduced so far allows permission transfer between threads only when threads are forked or joined. However, access to shared data such as a shared buffer requires various threads to obtain and relinquish permissions while the threads are running. Access to shared data is synchronized using *monitors*, which are guarded by mutual-exclusion locks. In Chalice, monitors can hold access permissions, just like activation records. Therefore, a thread can pass permissions to another thread by first storing them in a monitor, which allows the other thread to obtain them from the monitor.

When a thread wants to read the shared memory location, it competes for the mutual-exclusion lock using an **acquire** operation. Upon successful acquisition of the lock, the permissions stored in the monitor are transferred to the acquiring thread. When the thread is done writing, it uses the **release** operation to release the lock and transfer the permissions back into the monitor.

4.0 Monitor Invariants

In Chalice, like in C# and Java, any object can be used as a monitor. Each class has an associated *monitor invariant*, which describes the permissions stored in the monitors associated with the objects of this class. For instance, class `RockBand` in Fig. 3 declares a monitor invariant that says that the monitor of each `RockBand` object `rb` holds 100%

permission to `rb.memberCount`. These permissions are stored in the monitor when no thread holds the lock. The monitor invariant also contains a pure assertion stating that `memberCount` holds a non-negative value.

4.1 Object Life Cycle

An allocated object in Chalice can be in one of three states: *not-a-monitor*, *available*, and *held*. In the *not-a-monitor* state, which is the state of a newly allocated object, the object cannot be used as a monitor. That is, the **`acquire`** and **`release`** operations cannot be applied to the object and the monitor invariant need not hold. The **`share`** statement associates a monitor with such an object, transitioning the object from the *not-a-monitor* state to the *available* state. In the *available* state, the monitor holds the permissions indicated by the monitor invariant. The **`acquire`** statement transitions an object from the *available* state to the *held* state, and **`release`** transitions the object back to *available*.

To illustrate the object life cycle, consider the following code:

```
var rb := new RockBand { memberCount := 0 };
share rb;
// ...
acquire rb;
rb.AddMembers(1);
release rb;
var z := rb.memberCount;  // error: insufficient permission
                                  to read rb.memberCount
```

The **`share`** statement makes the object available, checks that the monitor invariant holds, and transfers the permissions entailed by the monitor invariant to `rb`'s monitor. This leaves the calling thread with 0% permission of `rb.memberCount`. Consequently, the thread cannot access this location.

Upon completion of the **`acquire`** statement, the monitor's permissions are transferred to the thread. The thread will have full access to `rb.memberCount` after the **`acquire`**. In particular, this gives it the permission to perform the method call. In the *available* state, the monitor invariant holds, and thus the monitor invariant can be assumed to hold just after the **`acquire`** statement.

Like the **`share`** statement, the **`release`** statement checks the monitor invariant to hold and then transfers the appropriate permission from the thread to the monitor. In the example, the monitor invariant is known to hold, as can be established from the second postcondition of `AddMembers` and the fact that the monitor invariant holds upon completion of the **`acquire`** statement.

The example further shows that `rb.memberCount` is not readable after the **`release`**.

5 Deadlock Prevention

To ensure mutual exclusion, the **`acquire`** operation suspends the execution of the acquiring thread until the requested monitor can be given to that thread. A well-behaved

program makes sure that other threads will eventually make such a monitor available. One possible program error is a *deadlock*, which occurs when a set of threads are suspended, each waiting for a monitor that is held by another thread in the set.

5.0 Locking Order

Chalice prevents deadlocks by breaking cycles among acquiring threads. This is done by letting a program associate each monitor with a *locking level* and then checking that the program acquires the monitors in ascending order. The locking levels are values from a set `Mu`, whose strict partial order is denoted <<. A program specifies the locking level of a monitor using the **`share`** statement, which takes an optional **`between ... and ...`** clause. Alternatively, a clause **`above ...`** or **`below ...`** may be used if only one bound is given. By default, the **`share`** statement uses **`above maxlock`**, where **`maxlock`** denotes the highest monitor currently held by the thread.

For example, consider a program fragment that allocates two objects, `a` and `b`, and associates them with monitors:

```
var a := new T;
share a above maxlock;
var b := new T;
share b above a;
```

Here, `a` is given a locking level above what the thread already holds, and `b` is given a locking level above that. Acquiring both of these monitors must be done in ascending order, so a thread must first acquire `a` and then `b`. An attempt at acquiring them in the opposite order results in the program verifier reporting an error:

```
acquire b;
acquire a;  // error: target lock might not be above maxlock
```

`release` operations can be done in any order.

The locking level of an object is recorded in a special field called `mu`. Access to this field is protected by permissions, just as for other fields, but the field cannot be changed by assignment statements.

Figure 7 shows an example program, the famous Dining Philosophers [12], here with 3 philosophers. A philosopher requires two forks to eat,[3] and these are passed as parameters to the `Run` method. The `Run` method's precondition specifies the relative locking order required of the parameters. Note the use of **`maxlock`**, the ordering <<, and the field `mu`. Since the precondition mentions `a.mu` and `b.mu`, the precondition must also require access to these fields; the example does so using two **`rd`** predicates. The example program avoids deadlocks by assigning forks to philosophers in an asymmetric way, as shown in the parameters passed to `Run` in the main program's **`fork`** statements.

[3] These forks denote eating utensils and have no relation to the **`fork`** statement.

```
class Fork { }

class Philosopher {
  method Run(a: Fork, b: Fork)
    requires rd(a.mu) && rd(b.mu);
    requires maxlock << a.mu && a.mu << b.mu;
  {
    while (true)
      invariant rd(a.mu) && rd(b.mu);
      invariant maxlock << a.mu && a.mu << b.mu;
    {
      // think...
      acquire a;
      acquire b;
      // eat...
      release a;
      release b;
    }
  }
}

class Program {
  method Main() {
    // create forks
    var f0 := new Fork;  share f0;
    var f1 := new Fork;  share f1 above f0.mu;
    var f2 := new Fork;  share f2 above f1.mu;

    // create philosophers
    var aristotle := new Philosopher;
    var plato := new Philosopher;
    var kant := new Philosopher;

    // start eating
    fork aristotle.Run(f0, f1);
    fork plato.Run(f1, f2);
    fork kant.Run(f0, f2);
  }
}
```

Fig. 7. Dining philosophers without deadlocks.

5.1 Example: Producer-Consumer

The producer-consumer problem is a classical example that illustrates the need for synchronization in programs with multiple threads and shared data. As an example, consider the program from Figs. 8 and 9. In this program, two worker threads concurrently access a shared buffer. One worker (called the producer) adds items to the buffer, while the other (called the consumer) removes items. To prevent data races, both worker threads must acquire the monitor of the shared buffer before reading or writing its fields.

Let's take a look at the code and the annotations. The first class of Fig. 8 is `Buffer`. `Buffer`'s monitor invariant indicates that the monitor of a `Buffer` object `b` protects the field `b.contents`. The field `contents` holds a sequence of integers, which can be of any length; thus, each `Buffer` object implements an unbounded buffer. The classes `Producer` and `Consumer` are similar to each other. Both `Produce` and `Consume` require that the given buffer can be acquired, by requiring read access to the buffer's `mu` field. In the body of a loop, both methods acquire the buffer to gain access to and update the field `buff.contents`. Note that a loop invariant must be provided which states that `buff` remains available. Finally, consider the `Main` method in Fig. 9. After creation of `buffer`, the main method has permission to access `buffer.contents`. The **`share`** operation consumes this permission and stores it in the monitor, but the thread still has permission to access `buffer.mu`. The preconditions of `Producer.Produce` and `Consumer.Consume` indicate that upon forking the worker threads a fraction (ϵ) of the permission to `buffer.mu` is passed to the workers. The conjunct **`maxlock`** `<< buff.mu` is trivially satisfied since a thread initially holds no monitors.

Since we use an unbounded buffer, the producer is always able to add new items to the buffer. However, when the buffer is empty, the consumer is not able to take out an item and has to wait until an item becomes available. In our example, we use busy-waiting to check repeatedly whether the buffer is still empty. A more efficient solution would use condition variables, which allow the consumer to wait until the producer signals the availability of new items. However, condition variables are not yet available in Chalice.

It is instructive to consider a variation of `Consume` where, instead of enclosing the assignments by an **`if`** statement, they are preceded by a loop

```
while (|buff.contents| == 0) { }
```

Such an implementation might fail to make *progress*, because the consumer might acquire the empty buffer and would then enter the loop without releasing the monitor. In that case, the producer could not acquire the monitor to add a new item, the buffer would stay empty, and the consumer would loop forever. Chalice does not prevent livelocks and does not check for termination. So our hypothetical program would verify.

6 Data Abstraction

A method contract specifies which permissions a method requires or returns, together with restrictions on the values of the corresponding locations. For example, the precondition of `RockBand.AddMembers` from Fig. 3 indicates that `AddMembers` requires

```
class Buffer {
  var contents: seq<int>;
  invariant acc(contents);
}

class Producer {
  method Produce(buff: Buffer)
    requires rd(buff.mu) && maxlock << buff.mu;
  {
    var x := 0; var y := 1;
    while (true)
      invariant rd(buff.mu) && maxlock << buff.mu;
    {
      lock (buff) {
        buff.contents := buff.contents ++ [x];
        var tmp := x; x := y; y := tmp + x;
      }
    }
  }
}

class Consumer {
  method Consume(buff: Buffer)
    requires rd(buff.mu) && maxlock << buff.mu;
  {
    while (true)
      invariant rd(buff.mu) && maxlock << buff.mu;
    {
      lock (buff) {
        if (0 < |buff.contents|) {
          var x := buff.contents[0];
          buff.contents := buff.contents[1..];
        }
      }
    }
  }
}
```

Fig. 8. A producer and a consumer. A **lock** (c) block is a syntactic shorthand for **acquire** c; *block;* **release** c.

```
class Program {
  method Main() {
    var buffer := new Buffer { contents := [] };
    share buffer;

    var producer := new Producer;
    fork producer.Produce(buffer);

    var consumer := new Consumer;
    fork consumer.Consume(buffer);
  }
}
```

Fig. 9. A client for the producer and consumer of Fig. 8.

permission to access `this.memberCount`. The method's postcondition ensures that this permission is returned to the callee and that the value of the corresponding location has been increased by `howMany`. While this specification is correct, it is not implementation independent, since it exposes the internal field `memberCount`. Exposing the internal representation is problematic for several reasons. First, exposing a class' internals tightly *couples* client code to the implementation. That is, a change to the implementation (for example, one might keep a sequence of band members instead of just an integer) would necessitate a modification of the method contract. Whenever a method contract is changed, the correctness of all its clients must be reverified. Second, if `RockBand` were an interface,[4] then the field `memberCount` would not be available and it would be impossible to specify `AddMembers`.

In this section, we explain how methods can be specified without exposing the class' internal representation. As we have mentioned before, pre- and postconditions consist of a number of pure assertions and a number of access assertions. Pure assertions can be abstracted over via *functions* (see Section 6.0), and both access assertions and pure assertions can be abstracted via *predicates* (see Section 6.1).

6.0 Functions

Functions are a way to abstract over the values of memory locations. More specifically, a function is an abbreviation for an expression with a precondition. For example, consider the new version of the class `RockBand` in Fig. 10. This class defines a function `getMemberCount` whose body is an integer expression. The function is defined to yield the value of the field `memberCount`. `getMemberCount`'s precondition indicates that `getMemberCount` can be applied only if the caller has permission to read `memberCount`. Without this precondition, `getMemberCount`'s definition is invalid as reading `memberCount` requires an access permission. Functions cannot change the program state. In particular, they can neither produce nor consume permissions. For that reason, it is not necessary

[4] Interfaces, abstract classes, and subclasses are currently not supported in Chalice.

```
class RockBand {
  var memberCount: int;
  invariant acc(memberCount) && 0 <= getMemberCount();

  function getMemberCount(): int
    requires rd(memberCount);
  { memberCount }

  method Init()
    requires acc(memberCount);
    ensures acc(memberCount) && getMemberCount() == 0;
  { memberCount := 0; }

  method AddMembers(howMany: int)
    requires acc(memberCount) && 0 <= howMany;
    ensures acc(memberCount);
    ensures getMemberCount() == old(getMemberCount()) + howMany;
  { memberCount := memberCount + howMany; }
}
```

Fig. 10. The class `RockBand` from Fig. 3 with a function to abstract over the value of the field `memberCount`. This version does not abstract over the use of `memberCount` in access predicates.

for functions to explicitly return permissions via postconditions. Instead, a function's caller automatically regains access to all permissions requested by the precondition when the function finishes.

The new version of `RockBand` does not explicitly dereference `memberCount` in method contracts, but uses the function `getMemberCount` instead. A function's meaning (that is, its body) is visible only to the module defining the function. But how can client code outside the module deduce which memory locations affect the function's return value? The answer is that clients can deduce this information from a function's precondition: a function can depend on a memory location `o.f` only if its precondition demands permission to access `o.f`. For example, `getMemberCount`'s value depends only on the field `memberCount`. Therefore, if a client changes some memory locations that do not include `memberCount`, it can safely assume that this heap update does not affect the result of `getMemberCount()`.

Although `RockBand`'s contracts in Fig. 10 no longer dereference `memberCount`, the specification is still not implementation independent, since it does not hide what permissions are required by method implementations. In the next subsection, we introduce predicates and demonstrate how they achieve complete implementation independence.

6.1 Predicates

Predicates are a way to abstract over permissions as well as values of memory locations. More specifically, a predicate is an abbreviation for a number of access assertions and

```
class RockBand {
  var memberCount: int;
  invariant valid;

  predicate valid
  { acc(memberCount) && 0 <= memberCount }

  function getMemberCount(): int
    requires valid;
  { unfolding valid in memberCount }

  method Init()
    requires acc(this.*);
    ensures valid && getMemberCount() == 0;
  { memberCount := 0; fold valid; }

  method AddMembers(howMany: int)
    requires valid && 0 <= howMany;
    ensures valid;
    ensures getMemberCount() == old(getMemberCount()) + howMany;
  { unfold valid; memberCount := memberCount + howMany; fold valid; }
}
```

Fig. 11. The class `RockBand` with implementation-independent contracts. The **`unfolding`** expression in function `getMemberCount` unfolds the predicate `valid` before evaluating the nested expression, `memberCount`. This unfolding is necessary to obtain the permission for this field.

a number of pure assertions. For example, the predicate `valid` in Fig. 11 is a shorthand for the permission to access `memberCount` and the condition that it holds a non-negative value. That is, the assertion `b.valid` is equivalent to **`acc`**`(b.memberCount) && 0 <= b.memberCount`. In Fig. 11, `RockBand`'s specification uses `valid` instead of explicitly requesting permission to access `memberCount`.

Each program point has one of two views of a predicate: the *abstract view* (or, folded view), which is independent of the predicate's definition, and the *concrete view* (or, unfolded view), where the predicate has been expanded according to its definition. Contrary to the functions described in the previous subsection, whose definitions are automatically available anywhere inside the module, a program switches between the two views of a predicate via two statements, **`fold`** and **`unfold`**. More specifically, **`fold`** `q` checks that `q`'s definition holds, consumes the permissions requested by that definition, and returns `q`. That is, one can think of **`fold`** `valid` as an invocation of a method with contract **`requires`** **`acc`**`(memberCount) && 0 <= memberCount;` **`ensures`** `valid`. Vice versa, the statement **`unfold`** `q` requires `q` and returns `q`'s definition. That is, one can think of **`unfold`** `valid` as an invocation of a method with contract **`requires`** `valid;` **`ensures`** **`acc`**`(memberCount) && 0 <= memberCount`.

Similarly to pre- and postconditions, predicate definitions must be well-defined. In particular, any dereference `o.f` in the definition of a predicate must be preceded by an

access predicate like **acc**(o.f). This means that a predicate cannot constrain the value of a memory location unless the predicate includes permission to access that location.

Note that the specification of RockBand in Fig. 11 is implementation independent. As a consequence, we can change RockBand's internal representation (within the boundaries set by the method contract) without having to worry about reverifying client code. The conjunct **acc**(**this**.*) in the precondition of Init is a shorthand for requesting access to all fields of **this**, which means that Init does not have to mention what the fields of **this** are.

Chalice provides a number of (command-line) options to reduce annotation overhead. Firstly, the option -autoMagic reduces the size of method contracts, predicates, and invariants by inferring access assertions and other definedness conditions implicit in the explicitly given expressions. For example, this option allows one to write the predicate valid in the class Rockband as follows:

```
predicate valid
{ 0 <= memberCount }
```

In particular, the conjunct **acc**(memberCount) is inferred from the fact that the expression 0 <= memberCount accesses this field. Secondly, the option -defaults instructs Chalice to automatically fold and unfold predicates on entry and exit of methods. For example, all **unfold** and **fold** statements are inferred and can be removed from Fig. 11 if the -defaults option is used. Finally, when Chalice cannot find a particular predicate in the heap, it first tries to automatically close it before reporting an error to the user. This behavior can be activated using -autoFold. When automatic folding is active, Chalice infers the **fold** statements of Fig. 11.

6.2 Example: Aggregate Objects

An aggregate object is an object that uses other objects to represent its internal state. As an example, consider the class Stack in Fig. 12. Stack is implemented in terms of a class List (not shown here). The specification for Stack is idiomatic for aggregate objects. In particular, the valid predicate not only includes permission to access its own fields, but also demands that the representation object be valid.

Note that the specification of Stack is implementation independent, since it does not expose the fact that the class is implemented in terms of a List. A consequence of implementation independence is that we never have to reverify client code when we change Stack's internal representation.

6.3 Example: Linked Data Structures

Many programs make use of linked data structures, that is, objects whose fields refer to instances of the same class. The class Node of Fig. 13 is an example of such a data structure. Node objects represent binary trees, where left and right refer to the left and right subtree, respectively. Linked data structures are typically specified via recursive predicates. For example, the definition of isTree is defined in terms of itself: o.isTree holds when the thread has permission to access o.value, o.left, and o.right and

```
class Stack {
  var contents: List;

  predicate valid
  { acc(contents) && contents.valid }

  function size(): int
    requires valid;
  { unfolding valid in contents.length() }

  method Init()
    requires acc(this.*);
    ensures valid && size() == 0;
  { contents := new List; call contents.Init(); fold valid; }

  method Push(x: int)
    requires valid;
    ensures valid && size() == old(size()) + 1;
  { unfold valid; call contents.Add(x); fold valid; }
}
```

Fig. 12. The class `Stack`, which exemplifies typical aggregate objects.

`o.left` and `o.right` are null or valid subtrees. This example illustrates that predicates are useful not only for abstraction, but also for specifying recursive data structures.

The method `Sum` computes the sum of the values in the trees by recursively calling itself on its subtrees. Note that `Sum` computes the sum of the left subtree in a different thread. This is just an internal detail of the implementation which is not visible to client code.

7 Example: Owicki-Gries Counter

In 1976, Owicki and Gries [34] proposed an axiomatic approach for verifying properties of concurrent programs. One of the examples they considered is shown in Fig. 14. In the example, a main thread forks two worker threads that both increment a shared counter (in Fig. 14, `c.value` represents the shared counter). The goal of the example is to prove that the counter's value equals 2 when both worker threads finish. To prove this assertion, Owicki and Gries relied on auxiliary variables. More specifically, each worker thread has a contribution variable that holds the amount added to the shared counter by that thread. The monitor invariant of the shared counter says that the counter's value is the sum of all contributions.

Since both worker threads must be able to compete for and update the counter, `OGCounter`'s monitor (Fig. 14) stores 100% permission for `value`. However, this implies that no thread has any information about the counter when the monitor is available. In particular, the postconditions of the worker cannot express that the counter has

```
class Node {
  var value: int;
  var left: Node; var right: Node;

  predicate isTree {
    acc(value) && acc(left) && acc(right) &&
    (left != null ==> left.isTree) &&
    (right != null ==> right.isTree)
  }

  method Init()
    requires acc(this.*);
    ensures isTree;
  { left := null; right := null; fold isTree; }

  method Sum() returns (total: int)
    requires isTree;
    ensures isTree;
  {
    var tmp: int;

    unfold isTree;
    total := value;
    var tk: token<Node.Sum>;
    if (left != null) { fork tk := left.Sum(); }
    if (right != null) { call tmp := right.Sum(); total := total + tmp; }
    if (left != null) { join tmp := tk; total := total + tmp; }
    fold isTree;
  }
}
```

Fig. 13. A linked tree data structure, illustrating a recursive predicate and a recursive method.

been incremented, since such a specification would require at least read permission for `value`. We solve this dilemma by introducing two fields `c0` and `c1` that take the place of the contribution variables in Owicki and Gries's example. The key idea of the Chalice encoding is to store only read permissions (here, 30%) to the contribution variables in the monitor. So, each worker can get only a partial permission for the contribution variables by acquiring the monitor. The remainder of the permissions (70%) is transferred from and to the main thread when the worker threads are forked and joined. Because of this read permission, the postcondition of the workers is allowed to contain information about the values of the contribution variables. Together with the last conjunct of the monitor invariant, `c0 + c1 == value`, this information allows the main thread to determine what the value of the counter is.

Let's turn our attention to the implementation of the `Main` method. After sharing `c`, `Main` no longer has access to `c.value`. However, it retains 70% permission to the field `c.c0` and `c.c1`. The `Main` method passes these permissions to two worker threads that each acquires `c` and increments `c.value`. By acquiring `c`, each worker thread gains full access to either `c0` or `c1` (depending on the value of the parameter `b`) and can update its value accordingly. Since each worker thread retains 70% after releasing the monitor, it can pass information about its contribution variable to its caller via the postcondition. The main thread gets to assume the postcondition of the worker threads when it joins them, we may deduce that both `c.c0` and `c.c1` equal `1`. When the main thread acquires the counter, it also gets to assume the monitor invariant. In particular, it may assume that `c0 + c1 == value` and, thus, that the subsequent assertion holds.

8 Rely-Guarantee Reasoning

As explained in the previous section, a thread can keep track of the effect of other threads on shared data structures by using auxiliary variables. However, when the number of threads is not known at compile time, the use of such variables becomes less elegant. For that reason, Chalice provides another way of specifying and restricting the effect of threads on the shared state, namely two-state invariants. A *two-state monitor invariant* can constrain and relate two states, the current state and a previous state referred to using **`old`** expressions. For example, the monitor invariant of the class `Counter` of Fig. 15 contains the permission to access `x` and specifies that its value can only grow.

Like single-state monitor invariants, a two-state monitor invariant must be shown to hold whenever a thread releases a monitor. **`old`**`(e)` expressions occurring in the invariant then refer to the value of the expression `e` at the time the monitor was last acquired. For example, the **`release`** statement of the method `Inc` verifies because the new value of `x` is larger than the value at the time the monitor was acquired. Method `Dec` on the other hand does not verify, because the monitor invariant does not hold when the monitor is released. Whenever a thread acquires a monitor, the corresponding monitor invariant may be assumed. **`old`**`(e)` expressions occurring in the monitor invariant then refer to the value of the expression `e` at the time the monitor was last shared or released. For example, the assert in the `Main` method of Fig. 15 verifies because of `Counter`'s two-state invariant. In particular, it follows from the invariant that on entry to the second monitor, `x`'s value is at least `tmp0`.

```
class OGCounter {
  var value: int;
  ghost var c0: int; ghost var c1: int; // contribution variables

  invariant acc(value) && acc(c0, 30) && acc(c1, 30) && c0 + c1 == value;

  method Add(b: bool)
    requires rd(mu) && maxlock << mu;
    requires (b ==> acc(c0, 70)) && (!b ==> acc(c1, 70));
    ensures rd(mu);
    ensures b ==> acc(c0, 70) && c0 == old(c0) + 1;
    ensures !b ==> acc(c1, 70) && c1 == old(c1) + 1;
  {
    lock (this) {
      if (b) {
        c0 := c0 + 1;
      } else {
        c1 := c1 + 1;
      }
      value := value + 1;
    }
  }

  method Main() {
    var c := new OGCounter { value := 0, c0 := 0, c1 := 0 };
    share c;

    fork tk0 := c.Add(true);
    fork tk1 := c.Add(false);
    join tk0; join tk1;

    lock (c) {
      assert c.value == 2;
    }
  }
}
```

Fig. 14. A program that uses worker threads to increment a shared counter. The annotations let Chalice prove the **assert** statement in `Main`. The fields `c0` and `c1` are auxiliary variables that are needed only to verify the program. Therefore, they are declared as **ghost**, which suggests to the compiler the optimization of omitting them in the executable code.

```
class Counter {
  var x: int;
  invariant acc(x) && old(x) <= x;

  method Main() {
    var c := new Counter; share c;

    fork c.Inc();

    var tmp0: int; var tmp1: int;
    lock (c) { tmp0 := c.x; }
    lock (c) { tmp1 := c.x; }

    assert tmp0 <= tmp1;
  }

  method Inc()
    requires rd(mu) && maxlock << mu;
  { lock (this) { x := x + 1; } }

  method Dec()
    requires rd(mu) && maxlock << mu;
  { lock (this) { x := x - 1; } /* error */ }
}
```

Fig. 15. A class `Counter` illustrating the use of two-state invariants.

A two-state monitor invariant is well-defined only if it is reflexive and transitive. For example, Chalice rejects the following invariant as it is transitive, but not reflexive:

```
invariant acc(x) && old(x) < x;
```

If Chalice did not check well-definedness, the code shown below would verify, although the assert statement fails during the program's execution:

```
var c := new Counter { x := 0 };
share c;
acquire c;
assert 0 < c.x;
```

The two-state monitor invariants described above allow only a limited form of rely-guarantee reasoning [23]. In particular, the conditions that are guaranteed and relied upon are the same for all threads. For information on more advanced forms of rely-guarantee, we refer the interested reader to [10, 41, 14].

9 Conclusion

In these lecture notes, we explained Chalice from a user's perspective. Readers who are interested in finding out what goes on behind the scenes are encouraged to read our research paper [30]. Further information can be found in the publications that inspired us: Fractional permissions were introduced by Boyland [6], who developed a type system to check concurrent programs for data races. Separation logic [36, 35, 33, 37] used permissions for program verification. Checkers based on symbolic execution for separation logic include jStar [13], VeriFast [22], and Smallfoot [4]. Smans et al. [39] showed how the ideas of separation logic can be incorporated into a traditional verifier based on verification condition generation and first-order theorem proving. The rules for reasoning about monitors were taken from Hoare [19]. Similar approaches are used by Jacobs et al. [21] in the context of Spec# [3], by Cohen et al. [8] in the context of C, and by Gotsman et al. [16] and by Hobor et al. [20] in separation logic. Haack et al. [17, 18] demonstrate how the separation-logic approach can be extended to handle fork/join and reentrant locks. The functions described in Section 6 are similar to pure methods (see, e.g., [28, 38]). Chalice's predicates are based on Parkinson's abstract predicates [35], whose use is similar to the valid/state paradigm from ESC/Modula-3 [26, 31].

Another model that reuses method contract annotations for concurrency is SCOOP [0]. Other techniques for proving the absence of data races in concurrent programs include model checking [32] and ownership type systems [11, 5, 9].

Readers who have become excited about automatic verifiers might also want to find out about techniques and tools for sequential programs. Ownership-based verification [29], dynamic frames [24, 40, 27], and regional logic [1] are interesting techniques for specifying sequential imperative programs, in particular for solving the frame problem. The research community has developed various other program verifiers, including ESC/Java [15], Jahob [42], Spec# [3, 2], VCC [8], and Dafny [27]. The Java Modeling Language (JML) [25, 7] is the de-facto standard for specifying Java programs.

Chalice demonstrates that permissions can express many concurrency patterns, including concurrent reading, thread-local and shared objects, aggregate objects, linked object structures, and thread collaboration. Other features such as dynamic changes of the locking order are also supported, but not covered in this tutorial. The absence of data races and deadlocks, preservation of monitor invariants, and other interesting properties can be verified automatically. Nevertheless, many concurrency concepts are not yet supported by Chalice; among them are condition variables (like Java's `wait-notify` mechanism), volatile variables, interlocked operations, and full rely-guarantee reasoning. We hope that the FOSAD summer school, and especially these lecture notes, will encourage readers to address some of these topics. With the pending open-source release of Chalice, they will get the opportunity to experiment with the tool and contribute their solutions.

Acknowledgments Jan Smans is a research assistant of the Fund for Scientific Research – Flanders (FWO). This research is partially funded by the Interuniversity Attraction Poles Programme Belgian State, Belgian Science Policy. Bart Jacobs showed us how to use permissions to encode the Owicki-Gries counter in Fig. 14.

References

0. Volkan Arslan, Patrick Eugster, Piotr Nienaltowski, and Sebastien Vaucouleur. SCOOP—concurrency made easy. In Jürg Kohlas, Bertrand Meyer, and André Schiper, editors, *Dependable Systems: Software, Computing, Networks, Research Results of the DICS Program*, volume 4028 of *Lecture Notes in Computer Science*, pages 82–102. Springer, 2006.
1. Anindya Banerjee, David A. Naumann, and Stan Rosenberg. Regional logic for local reasoning about global invariants. In Jan Vitek, editor, *ECOOP 2008 — Object-Oriented Programming, 22nd European Conference*, volume 5142 of *Lecture Notes in Computer Science*, pages 387–411. Springer, July 2008.
2. Mike Barnett, Bor-Yuh Evan Chang, Robert DeLine, Bart Jacobs, and K. Rustan M. Leino. Boogie: A modular reusable verifier for object-oriented programs. In Frank S. de Boer, Marcello M. Bonsangue, Susanne Graf, and Willem-Paul de Roever, editors, *Formal Methods for Components and Objects: 4th International Symposium, FMCO 2005*, volume 4111 of *Lecture Notes in Computer Science*, pages 364–387. Springer, 2006.
3. Mike Barnett, K. Rustan M. Leino, and Wolfram Schulte. The Spec# programming system: An overview. In Gilles Barthe, Lilian Burdy, Marieke Huisman, Jean-Louis Lanet, and Traian Muntean, editors, *CASSIS 2004, Construction and Analysis of Safe, Secure and Interoperable Smart Devices*, volume 3362 of *Lecture Notes in Computer Science*, pages 49–69. Springer, 2005.
4. Josh Berdine, Cristiano Calcagno, and Peter W. O'Hearn. Smallfoot: Modular automatic assertion checking with separation logic. In Frank S. de Boer, Marcello M. Bonsangue, Susanne Graf, and Willem-Paul de Roever, editors, *Formal Methods for Components and Objects (FMCO 2005)*, volume 4111 of *Lecture Notes in Computer Science*, pages 115–137. Springer, 2006.
5. Chandrasekhar Boyapati, Robert Lee, and Martin Rinard. Ownership types for safe programming: Preventing data races and deadlocks. In *Proceedings of the 2002 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications, OOPSLA 2002*, pages 211–230. ACM, November 2002.
6. John Boyland. Checking interference with fractional permissions. In Radhia Cousot, editor, *Static Analysis, 10th International Symposium, SAS 2003*, volume 2694 of *Lecture Notes in Computer Science*, pages 55–72. Springer, June 2003.
7. Lilian Burdy, Yoonsik Cheon, David R. Cok, Michael D. Ernst, Joeseph R. Kiniry, Gary T. Leavens, K. Rustan M. Leino, and Erik Poll. An overview of JML tools and applications. *International Journal on Software Tools for Technology Transfer*, 7(3):212–232, June 2005.
8. Ernie Cohen, Markus Dahlweid, Mark Hillebrand, Dirk Leinenbach, Michal Moskał, Thomas Santen, Wolfram Schulte, and Stephan Tobies. VCC: A practical system for verifying concurrent C. In *Theorem Proving in Higher Order Logics, 21st International Conference, TPHOLs 2009*, volume 5674 of *Lecture Notes in Computer Science*, August 2009.
9. David Cunningham, Sophia Drossopoulou, and Susan Eisenbach. Universe types for race safety. In *Proceedings of the 1st Workshop on Verification and Analysis of Multi-threaded Java-like Programs (VAMP)*, number ICIS-R07021 in Technical Report, pages 20–51. Radboud University Nijmegen, September 2007.
10. Willem-Paul de Roever, Frank de Boer, Ulrich Hanneman, Jozef Hooman, Yassine Lakhnech, and Job Zwiers. *Concurrency Verification*. Cambridge University Press, 2001.
11. Werner Dietl, Sophia Drossopoulou, and Peter Müller. Generic universe types. In Erik Ernst, editor, *ECOOP 2007 — Object-Oriented Programming, 21st European Conference*, volume 4609 of *Lecture Notes in Computer Science*, pages 28–53. Springer, July–August 2007.
12. Edsger W. Dijkstra. Hierarchical ordering of sequential processes. *Acta Informatica*, 1:115–138, 1971.

13. Dino Distefano and Matthew J. Parkinson. jStar: Towards practical verification of Java. In Gail E. Harris, editor, *Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*, volume 37(11) of *SIGPLAN Notices*, pages 213–226. ACM, October 2008.
14. Xinyu Feng. Local rely-guarantee reasoning. In *Proceedings of the 36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2009*, pages 315–327. ACM, January 2009.
15. Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. Extended static checking for Java. In *Proceedings of the 2002 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 234–245. ACM, May 2002.
16. Alexey Gotsman, Josh Berdine, Byron Cook, Noam Rinetzky, and Mooly Sagiv. Local reasoning for storable locks and threads. In Zhong Shao, editor, *Programming Languages and Systems, 5th Asian Symposium, APLAS 2007*, volume 4807 of *Lecture Notes in Computer Science*, pages 19–37. Springer, November–December 2007.
17. Christian Haack, Marieke Huisman, and Clément Hurlin. Reasoning about Java's reentrant locks. In G. Ramalingam, editor, *Programming Languages and Systems, 6th Asian Symposium, APLAS 2008*, volume 5356 of *Lecture Notes in Computer Science*, pages 171–187. Springer, December 2008.
18. Christian Haack and Clément Hurlin. Separation logic contracts for a Java-like language with fork/join. In José Meseguer and Grigore Rosu, editors, *Algebraic Methodology and Software Technology, 12th International Conference, AMAST 2008*, volume 5140 of *Lecture Notes in Computer Science*, pages 199–215. Springer, July 2008.
19. Tony Hoare. Monitors: An operating system structuring concept. *Communications of the ACM*, 17(10):549–557, 1974.
20. Aquinas Hobor, Andrew W. Appel, and Francesco Zappa Nardelli. Oracle semantics for concurrent separation logic. In Sophia Drossopoulou, editor, *Programming Languages and Systems, 17th European Symposium on Programming, ESOP 2008*, volume 4960 of *Lecture Notes in Computer Science*, pages 353–367. Springer, March–April 2008.
21. Bart Jacobs, K. Rustan M. Leino, Frank Piessens, Wolfram Schulte, and Jan Smans. A programming model for concurrent object-oriented programs. *ACM Transactions on Programming Languages and Systems*, 31(1), December 2008.
22. Bart Jacobs and Frank Piessens. The VeriFast program verifier. Technical Report CW-520, Department of Computer Science, Katholieke Universiteit Leuven, August 2008.
23. Cliff B. Jones. Specification and design of (parallel) programs. In *Proceedings of IFIP'83*, pages 321–332. North-Holland, 1983.
24. Ioannis T. Kassios. Dynamic frames: Support for framing, dependencies and sharing without restrictions. In Jayadev Misra, Tobias Nipkow, and Emil Sekerinski, editors, *FM 2006: Formal Methods, 14th International Symposium on Formal Methods*, volume 4085 of *Lecture Notes in Computer Science*, pages 268–283. Springer, August 2006.
25. Gary T. Leavens, Albert L. Baker, and Clyde Ruby. JML: A notation for detailed design. In Haim Kilov, Bernhard Rumpe, and Ian Simmonds, editors, *Behavioral Specifications of Businesses and Systems*, pages 175–188. Kluwer Academic Publishers, 1999.
26. K. Rustan M. Leino. *Toward Reliable Modular Programs*. PhD thesis, California Institute of Technology, 1995. Technical Report Caltech-CS-TR-95-03.
27. K. Rustan M. Leino. Specification and verification of object-oriented software. In *Engineering Methods and Tools for Software Safety and Security*, volume 22 of *NATO Security Through Science Series; Sub-Series D*, pages 231–266. IOS Press, 2009. Marktoberdorf 2008 lecture notes.
28. K. Rustan M. Leino and Ronald Middelkoop. Proving consistency of pure methods and model fields. In Marsha Chechik and Martin Wirsing, editors, *Fundamental Approaches to*

Software Engineering, 12th International Conference, FASE 2009, volume 5503 of *Lecture Notes in Computer Science*, pages 231–245. Springer, March 2009.
29. K. Rustan M. Leino and Peter Müller. Object invariants in dynamic contexts. In Martin Odersky, editor, *ECOOP 2004 — Object-Oriented Programming, 18th European Conference*, volume 3086 of *Lecture Notes in Computer Science*, pages 491–516. Springer, June 2004.
30. K. Rustan M. Leino and Peter Müller. A basis for verifying multi-threaded programs. In Giuseppe Castagna, editor, *Programming Languages and Systems, 18th European Symposium on Programming, ESOP 2009*, volume 5502 of *Lecture Notes in Computer Science*, pages 378–393. Springer, March 2009.
31. K. Rustan M. Leino and Greg Nelson. Data abstraction and information hiding. *ACM Transactions on Programming Languages and Systems*, 24(5):491–553, September 2002.
32. Madanlal Musuvathi, Shaz Qadeer, Thomas Ball, Gerard Basler, Piramanayagam Arumuga Nainar, and Iulian Neamtiu. Finding and reproducing Heisenbugs in concurrent programs. In Richard Draves and Robbert van Renesse, editors, *8th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2008*, pages 267–280. USENIX Association, December 2008.
33. Peter W. O'Hearn. Resources, concurrency, and local reasoning. *Theoretical Computer Science*, 375(1–3):271–307, 2007.
34. Susan Owicki and David Gries. Verifying properties of parallel programs: An axiomatic approach. *Communications of the ACM*, 19(5):279–285, May 1976.
35. Matthew J. Parkinson and Gavin M. Bierman. Separation logic and abstraction. In Jens Palsberg and Martín Abadi, editors, *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2005*, pages 247–258. ACM, January 2005.
36. Matthew John Parkinson. *Local Reasoning for Java*. PhD thesis, University of Cambridge, 2005.
37. John C. Reynolds. Separation logic: A logic for shared mutable data structures. In *17th IEEE Symposium on Logic in Computer Science (LICS 2002)*, pages 55–74. IEEE, July 2002.
38. Arsenii Rudich, Ádám Darvas, and Peter Müller. Checking well-formedness of pure-method specifications. In Jorge Cuéllar, Tom Maibaum, and Kaisa Sere, editors, *FM 2008: Formal Methods, 15th International Symposium on Formal Methods*, volume 5014 of *Lecture Notes in Computer Science*, pages 68–83. Springer, May 2008.
39. Jan Smans, Bart Jacobs, and Frank Piessens. Implicit dynamic frames: Combining dynamic frames and separation logic. In *ECOOP 2009 — Object-Oriented Programming, 23rd European Conference*, July 2009.
40. Jan Smans, Bart Jacobs, Frank Piessens, and Wolfram Schulte. An automatic verifier for Java-like programs based on dynamic frames. In José Luiz Fiadeiro and Paola Inverardi, editors, *Fundamental Approaches to Software Engineering, 11th International Conference, FASE 2008*, volume 4961 of *Lecture Notes in Computer Science*, pages 261–275. Springer, March–April 2008.
41. Viktor Vafeiadis and Matthew Parkinson. A marriage of rely/guarantee and separation logic. In Luís Caires and Vasco Thudichum Vasconcelos, editors, *CONCUR 2007 — Concurrency Theory, 18th International Conference*, volume 4703 of *Lecture Notes in Computer Science*, pages 256–271. Springer, September 2007.
42. Karen Zee, Viktor Kuncak, and Martin C. Rinard. Full functional verification of linked data structures. In Rajiv Gupta and Saman P. Amarasinghe, editors, *Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation*, pages 349–361. ACM, June 2008.