Modeling Concurrency in Dafny

K. Rustan M. Leino[1,2(✉)]

[1] Microsoft Research, Redmond, WA, USA
leino@acm.org
[2] Imperial College London, London, UK

Abstract. This article gives a tutorial on how the Dafny language and verifier can be used to model a concurrent system. The running example is a simple ticket system for mutual exclusion. Both safety and, under the assumption of a fair scheduler, liveness are verified.

1 Introduction

Knowing how to reason about the correctness of programs is an important skill for every software engineer. In some situations, functional correctness is so important that it makes sense to reason formally. This is done by being precise about the *pre- and postcondition specifications* of procedures, the *invariants* of data structures, and the justifications for why the program upholds these specifications and invariants. Today's tools make it possible to carry out and check this reasoning mechanically. In other situations, functional correctness is deemed less critical. Still, the concepts of pre- and postconditions and invariants apply and guide good program design, even if these are not checked rigorously in every possible execution of the program.

In my lectures at SETSS 2017, I taught program-correctness concepts like pre- and postconditions and invariants through many examples. I used the programming language Dafny [18,20], which includes specification constructs and has an automated program verifier that checks that programs meet their specifications. The literature already contains some tutorials that use Dafny for this purpose [6,11,14,17,19]. So, instead of repeating those tutorials, I am using these lecture notes to explain a more advanced use of Dafny, namely the modeling of a simple concurrent algorithm in Dafny. The reasoning that I show here is representative of how several popular formal systems reason about the behavior of concurrent threads of execution. A benefit you get from doing the modeling in Dafny is the automated verification.

If you are looking to learn to do this kind of modeling on your own, I suggest you follow along by entering the Dafny declarations into a buffer in the Dafny IDE for VS Code, Visual Studio, or Emacs. You can install Dafny and an IDE plug-in from https://github.com/Microsoft/dafny/. Alternatively, you can enter the declarations in your web browser at http://rise4fun.com/dafny, periodically clicking on the Play button to let Dafny check your program for syntactic and semantic correctness.

J. P. Bowen et al. (Eds.): SETSS 2017, LNCS 11174, pp. 115–142, 2018.
https://doi.org/10.1007/978-3-030-02928-9_4

2 Concurrency

A concurrent system consists of a number of processes that each executes some code. If we think of the execution of a process as a sequence of tiny steps, then we can model the entire system as an *interleaving* of these tiny-step sequences. To make sure our model captures every possible behavior of the concurrent system, it is important that each tiny step is so tiny that we can think its execution as taking place all by itself, impossibly interrupted or interfered with by the execution of other tiny steps. We call such a tiny step an *atomic event* (or *atomic step*, or *atomic action*). In a typical model, atomic events are limited to one read or write of a shared variable.

To model a concurrent system, we consider what possible atomic events may take place during the running of the system. This is the common idea in formalisms and tools like Chandy and Misra's UNITY [5], Back and Sere's action systems [4], Abrial's Event-B [1] (implemented in the Rodin tool [3]), and Lamport's TLA+ [15]. As designers of a model, we decide what each atomic event stands for. By choosing an appropriate level of abstraction for the events, we can use the same modeling techniques when writing a high-level specification as when writing a low-level implementations. To see how this kind of thinking can lead to nice specifications, see for example Leslie Lamport's TLA+ lectures [16] or Jean-Raymond Abrial's Event-B lectures [2].

Code executed by a process is authored as a program where control flow is implicit. To model such code, we introduce an explicit control state for each process. In the extreme, the control state can be a program counter, but many times we don't need to be as concrete as that.

Dafny is a programming language designed to support formal reasoning [18]. Even though the language is sequential, we can use it to model concurrent systems in the way I just alluded to. For instance, we can write each event as a little body of code, and we can give such a piece of code a name by declaring it as a procedure, which in Dafny is known as a *method*. An approach like this makes use of some programming constructs in Dafny, but mostly, it uses Dafny as a logical foundation for the modeling.

To illustrate this approach, I will describe a simple ticket system for mutual exclusion and model this in Dafny. I will use two different models. The first model defines atomic events by methods that change the program state. The second model defines atomic events by predicates over pairs of consecutive states. The main theorem to be proved about the ticket system is that, at any one time, at most one process is in its critical section. This is called a *safety property* and can be proved in either model. Another theorem of interest is that a process in the ticket system never gets stuck. This is called a *liveness property* and is most easily described in the second model. More complex applications of Dafny to specifying and implementing concurrent systems are found in the IronFleet project [10].

3 The Ticket System

The concurrent system we are going to reason about can be described as follows.

A fixed number of philosophers are gathered in a library. Mostly, they just sit around and think. Sometimes, a philosopher may become hungry and need to eat. Attached to the library is a kitchen. Unfortunately, the kitchen is undergoing some renovations, so it would be dangerous to have more than one person in the kitchen at any one time. To manage the contention for the shared kitchen resource, a ticket system has been installed.

The ticket system has a ticket dispenser that dispenses numbered tickets in increasing sequential order. There is no upper limit on these numbers. It is very quick to dispense a ticket, so we consider obtaining a ticket to be an atomic event.

The ticket system also includes a display that says "serving" and shows a number. The philosophers decide that only a philosopher whose ticket number agrees with the display is allowed into the kitchen. The button that advances the "serving" display to the next number is located in the kitchen. Upon leaving the kitchen, a philosopher presses the button to advance the number displayed.

4 Pseudo Code

Each philosopher operates as described by the following pseudo code:

```
forever repeat
{
  Thinking:
  // ...

  t, ticket := ticket, ticket + 1;  // Request
  Hungry:

  wait until serving = t;  // Enter
  Eating:
  // ...
  serving := serving + 1;  // Leave
}
```

A philosopher can be in one of three states: thinking, hungry, or eating. It starts in the thinking state. What exactly a philosopher is thinking about is not relevant to the algorithm, so the pseudo code abstracts over it.

To enter the hungry state, a philosopher grabs the next ticket, an operation I will refer to as `Request`. The ticket number obtained is recorded into a variable `t` that is local to the philosopher.

In the hungry state, a philosopher waits until the "serving" display shows `t`. The philosopher then enters the kitchen, which is represented as the eating state.

This `Enter` operation can take place only when the given condition holds. Alternatively, you can think of `Enter` as an operation that is attempted repeatedly, each attempt being one atomic event, until the given condition holds.

When a philosopher is done in the kitchen, it increments the "serving" display and returns back to the thinking state. I will refer to this operation as `Leave`. What exactly the philosopher prepares and eats in the kitchen is not relevant to the algorithm, so the pseudo code abstracts over it.

5 Formalizing the Ticket System

To formalize the ticket system in Dafny, we need to encode the ticket dispenser and the "serving" display. We also need a way to talk about each philosopher, which from now on I will call a *process*, along with the state of the process and any ticket it may be holding.

5.1 Processes as an Uninterpreted Type with Equality

To model the identity of processes, we introduce a type. In the most abstract way, we can do that by:

```
type Process
```

which introduces `Process` as the name of an uninterpreted type. Dafny allows us to be more concrete. For example, we can say that `Process` is just a synonym for the integers:

```
type Process = int
```

Or we can define `Process` to be an enumeration of a specific set of names, perhaps inspired by the names of our friends:

```
datatype Process = Agnes | Agatha | Germaine | Jack
```

In our case, we don't need more than an uninterpreted type. Almost. We do need to know that the type comes equipped with the ability to compare its values with equality. The notation for saying this is:

```
type Process(==)
```

(This `"(==)"` suffix is among the most cryptic of syntactic constructs in Dafny. It is unfortunate that, in this example, it's the first thing we need.)

5.2 Names of Process Control States

We need names for the program labels in our pseudo code. We define these in an enumeration type that we call `CState` (for *control state*):

```
datatype CState = Thinking | Hungry | Eating
```

5.3 Ticket System as a Class

Representing the ticket dispenser and "serving" display requires some state. For what I want to show in this article about the method formulation of events, it is simplest to declare this state as fields of a class. A class in Dafny is like a class in object-oriented languages, but there is nothing object-oriented in our example, so it is not particularly useful to think of our class like that. For our present purposes, all we need to know about a class is the following:

– A class can have state, which is introduced by `var` declarations inside the class. Borrowing from object-oriented terminology, these variables are sometimes called *fields*.
– A class can have methods, which are named bodies of code that operate on the state.
– A class has a *constructor* that initializes the state.
– A class can also declare *functions* and *lemmas*. I'll describe these when we need them.

In fact, for our purposes, there is not much difference between a class and what might be a module with some procedures and global variables. There are no global variables in Dafny, so that is why I use a class for our example.

The class itself is declared as follows:

```
class TicketSystem
{
}
```

All remaining declarations that I describe in this section and the next are to be placed inside the curly braces of this class declaration. Two of those declarations are for the ticket dispenser and "serving" display, each of which is represented by an integer:

```
var ticket: int
var serving: int
```

It will be convenient to have a name for the fixed set of processes, so we introduce a variable to store that set:

```
var P: set<Process>
```

In fact, since `P` will never change once it has been initialized, we can declare it to be an immutable field:

```
const P: set<Process>
```

Next, we introduce the state for each process. This state consists of a control-state value (that is, thinking, hungry, or eating) and an integer that denotes the value of the ticket held by the process. The ticket value is relevant only if the process is hungry or eating; when the process is thinking, we don't care what the value of this integer is.

But how do we introduce this state for each process? There's more than one way to do this in Dafny. We will use a way that perhaps would be most natural to someone familiar with TLA+, or for that matter, someone familiar with Alloy [13]. It is to use maps from processes to values. We introduce a map `cs` from each process in `P` to a control state and a map `t` from each process in `P` to an integer:

```
var cs: map<Process, CState>
var t: map<Process, int>
```

To look up the `CState` value for a process `p`, we consult the map `cs` using the expression `cs[p]`.

We need these maps only for processes in `P`, but (unlike in TLA+) you cannot use a state variable `P` as the domain of these maps. (Actually, in TLA+, `P` would not be a variable, but a logical constant.) That's why we declare the domain type of these maps to be `Process`, which includes not just the processes in our instance of the ticket system, but every conceivable process. This is no problem, because a `map` in Dafny is a (possibly) partial map, anyway. Stated differently, the domain of a `map` in Dafny is some subset of the values denoted by the domain type of the map. In other words, if you think of a map as a set of key-value pairs, then this set may or may not contain a key-value pair for every value of the map-domain type.

5.4 System Invariant

When we access one of our maps for a particular process, say `p`, we need to know that `p` is in the domain of the map (that is, that `p` is among the *keys* of the map). In our ticket system, it will always be the case that the domain of the maps is `P`. (Actually, we will never have occasion to access these maps outside `P`, so the important property is that `P` is a *subset* of the domain of each map.) We call this an *invariant* of the system, and (we will show that) it holds initially and is maintained by every atomic operation.

If we are given arbitrary values for our variables, then this invariant condition may or may not hold. We introduce a function that tells us whether or not the condition holds. We'll give this function the name `Valid`:

```
predicate Valid()
 reads this
{
 cs.Keys == t.Keys == P
}
```

There are several points to explain about the definition of `Valid`.

- A function that returns a boolean is synonymously known as a *predicate*. The `predicate` declaration is just a nice way to write `function Valid(): bool`.
- In Dafny, functions are allowed to depend on the program state, but they must declare which parts of the program state they depend on. This concept

is called *framing* (see, for example, [9]). For this purpose, a function (and also a predicate, since a predicate is just a function that returns a boolean) has a `reads` clause. Here, a bit of object-oriented notation creeps in: the keyword `this`. By declaring the function with `reads this`, we are saying that it may depend on the values of the fields declared in the class. Framing is not central to our problem here, but you have to include the `reads` clause or Dafny would complain that the function body illegally reads `cs` and `t`.

- The value of the function—here, of type boolean—is given by its body, which is an expression enclosed in the curly braces that follow the function signature and specification.
- Equality in Dafny uses the syntax so familiar to all C and Java programmers: `==`.
- In Dafny, a number of operators, including equality, are *chaining*. This means that you can string them together, just like you would in a mathematical textbook. In particular, `cs.Keys == t.Keys == P` is equivalent to the conjunction ("and") of `cs.Keys == t.Keys` and `t.Keys == P`. By the way, conjunction is written `&&` and disjunction ("or") is written `||`.
- The domain of a map is a set that is retrieved by the member `.Keys`.

(As I remarked above, we only need that `P` is a subset of the keys of each map. If we had chosen to formalize that, we would instead have written `P <= cs.Keys && P <= t.Keys` in the body of `Valid`. In Dafny, `<=` applied to sets denotes the subset relation.)

As we go along, we will find more things to add to the body of `Valid`. Remember, we think of `Valid()` as describing the invariant of our system, but as far as Dafny is concerned, `Valid` is just a function that sometimes returns `false` and sometimes returns `true`. What makes it appropriate to think of `Valid()` as an invariant has to do with the way we use it. We'll see this soon.

5.5 Initializing the Ticket System

We are now ready to write a little code to initialize the ticket system. This code is placed in a special method called a *constructor*. We parameterize the constructor by a set of processes.

```
constructor (processes: set<Process>)
 ensures Valid()
{
 P := processes;
 ticket, serving := 0, 0;
 cs := map p | p in processes :: Thinking;
 t := map p | p in processes :: 0;
}
```

The specification of the `TicketSystem` constructor has a *postcondition*, which is declared by an `ensures` clause. It expresses that, upon termination of the constructor, `Valid()` will hold. Dafny checks this, of course. In other words, this postcondition says that initialization establishes the invariant. For this reason, it is important to include a constructor—otherwise, we may accidentally write down an invariant that is equivalent to `false`, in which case all of the theorems we prove about the system will hold trivially.

The body of the constructor uses imperative programming statements that can change the state. In the second line, I am using a *parallel assignment*, because I think it looks nice. It would be equally good to write this as two separate assignments. Likewise, it would be equally good to write all five assignments as one parallel assignment.

The assignments to `cs` and `t` make use of map comprehensions. The first of these can be read as "the map from `p`, where `p` is a process in `processes`, to the value `Thinking`". Dafny infers the type of `p` to be `Process`.

In the map comprehension used to initialize the map `t`, we are being overly specific when we say that `t` maps each process to `0`. Since every process is initially thinking, this value is irrelevant. However, to establish the invariant, we do need to assign to `t` a map whose domain is `processes`, which our assignment achieves.

All concurrency formalisms have some way to describe the possible initial states of the system. For example, in UNITY, an **`initially`** clause gives a predicate that describes the possible initial states. In TLA+ and Event-B, too, the initial condition is phrased as a predicate.

At this point, we have declared all types and variables involved in the modeling of our system. We have also declared an invariant and provided a constructor that initializes the system to satisfy the invariant. What remains to be done is to formalize the atomic steps of the system. And, of course, to state and verify a correctness theorem about the system.

5.6 Specifying the Atomic Events

There are three atomic events for us to formalize: `Request`, `Enter`, and `Leave`. Actually, we need to formalize each of these for each process. To do that, we parameterize each atomic event by a process. As I have mentioned before, we will model each atomic event by a method. To parameterize the description of an atomic event, we simply declare the method to take a parameter.

The parameter, which I will name `p`, is of type `Process`. We only want to model atomic events for processes in our system, so we write a *precondition*, which is declared by a `requires` clause. Moreover, we are only interested in modeling behavior from states that satisfy the invariant, so we also include `Valid()` as a precondition of our methods.

An atomic event for a process is typically enabled only in certain circumstances. These circumstances depend on the control state of the process. For example, the atomic event `Request(p)` is enabled only when `p` is thinking. Therefore, we add a constraint on `cs[p]`, like `cs[p] == Thinking`, to the precondition.

It is important that each atomic event maintain the invariant. To express and check this property, we add `Valid()` as a postcondition of each method, just as for the constructor.

There is one more detail. Recall that a function must declare (in a `reads` clause) which parts of the program state it depends on. In a similar way, a method must declare (in a `modifies` clause) which parts of the program state it updates. By declaring our methods with `modifies this`, we are saying that they may update the fields in the class. This framing does not really play a role in our example, but we have to declare it or Dafny would complain that the method body illegally updates the fields.

In summary, each atomic operation in our example will be modeled by a method whose signature and specification look like this:

```
method AtomicStep(p: Process)
 requires Valid() && p in P && cs[p] == ...
 modifies this
 ensures Valid()
```

5.7 Implementing the Atomic Events

Having laid down the foundation, we can now easily define the three atomic events of our example. As you check these, you may want to compare them with the pseudo code above.

```
method Request(p: Process)
 requires Valid() && p in P && cs[p] == Thinking
 modifies this
 ensures Valid()
{
 t, ticket := t[p := ticket], ticket + 1;
 cs := cs[p := Hungry];
}
```

The notation `t[p := ticket]` is a map-update expression. It stands for a map that is like `t`, except that it maps `p` to `ticket`. (In TLA+, such an expression is written `[t` **`EXCEPT`** `![p] = ticket]`.) The body of `Request` thus records the current ticket as the ticket number held by `p`, increments the current ticket, and changes the control state of `p` to `Hungry`.

In the body of `Request`, I happen to be using one parallel assignment (updating both `t` and `ticket`) followed by one other assignment (updating `cs`), because I think this looks nice in the program text. However, there are many other sequences of assignments that achieve the same effect. By using a method to model an atomic event, the intermediate states inside the method body are not relevant, so the cosmetic choices we make inside the method body have no bearing on the model.

```
method Enter(p: Process)
 requires Valid() && p in P && cs[p] == Hungry
 modifies this
 ensures Valid()
```

```
    {
     if t[p] == serving {
       cs := cs[p := Eating];
     }
    }

  method Leave(p: Process)
     requires Valid() && p in P && cs[p] == Eating
     modifies this
     ensures Valid()
    {
     serving := serving + 1;
     cs := cs[p := Thinking];
    }
```

These three methods model the three atomic events in our system.

5.8 On the Atomicity of Events

To justify calling a method in our model an *atomic* event, we need to make sure that the method body is something that really can be completed without interference of other processes in the system we are modeling. Typically, this means that the body is allowed at most one read or write of one shared variable. Updating a map at a process `p` counts as an update that is local to `p`. In other words, we think of `cs[p]` as a local variable, not as a shared variable. In our example, two of our methods read or write a shared variable more than once, so why do we think our modeling is done properly? Let's take a closer look at these two methods.

Method `Request` reads `ticket` twice and writes it once, with the effect of incrementing `ticket` and recording its previous value. One way to achieve this in an actual implementation of our system would be to use a mutual-exclusion lock. But if we had such a lock, then we could use that lock as the way for a philosopher to enter the kitchen (that is, for a process to enter its critical section). The whole point of the ticket system is to *implement* a mutual-exclusion lock. Luckily, there are less heavy-handed ways than a lock to implement these accesses of `ticket`. For example, these accesses can be implemented by a hardware-supported atomic-increment operation, which is found in many modern machines. Such an operation increments the value stored in a memory location and at the same time returns its value from before the increment. With such an operation in mind, we feel justified in thinking about method `Request` as an atomic event.

The other method that reads or writes a shared variable more than once is `Leave`, which increments `serving`. We could justify this increment by a hardware atomic-increment operation as well, but for `Leave`, there is a less imposing justification. The `serving` variable is incremented only by the philosopher in the kitchen (that is, by the process in the critical section), so (if our ticket system does indeed provide mutual exclusion, then) `serving` is stable in `Leave`. That justifies calling `Leave` an atomic event without appealing to any hardware support.

In fact, there's yet one more way to deal with the atomicity of `Leave`. By the time a `Leave(p)` event takes place, the ticket held by `p` is equal to `serving`. Thus, instead of reading the shared variable `serving` as in the code above, we can read the local `t[p]`. If you want to check that what I'm saying is true, you can insert the following statement at the beginning of the body of `Leave`:

```
assert t[p] == serving;
```

In the program we have developed so far, the verifier will flag this assertion as an error, but once we're done with our correctness theorem below, the verifier will be able to verify it. (This assertion introduces yet one more read of the shared variable `serving`. But it is okay, because an assertion is to be considered as specification-only, so we don't need to worry about which variables it mentions.)

This completes our encoding of the atomic events in our model. What remains to do is to state and prove a correctness theorem that shows that the ticket system does indeed provide mutual exclusion.

5.9 Notes About Other Formalisms

In Event-B, the state transition of an event is given as a set of assignment statements. These assignments are performed in parallel, so the order in which you list them does not matter. In Dafny, the order of statements matters. However, whenever you use a parallel assignment, all right-hand sides are evaluated before any state change takes place. So, if you want an Event-B style body in Dafny, use one parallel assignment statement.

In UNITY, a state transition is also given as a parallel assignment. The syntax of the assignment is more like that in Dafny.

In TLA+, events are specified as two-state predicates, where a primed variable is used to refer to the value of the variable in the post-state. For example, the increment of `ticket` in `Request` is written `ticket' = ticket + 1` in TLA+. I will show such predicates in Sect. 8.3.

6 Correctness Theorem

The correctness of our ticket system comes down to showing that no two processes are ever in the eating state at the same time. To show that this condition holds in every state of the system, we prove a theorem that the invariant (which holds in every state) implies the condition. This can be stated in Dafny using a `lemma` declaration. A lemma in Dafny is really just a method, except that it will not be compiled into code. Like any method, a lemma has a pre- and postcondition. The precondition denotes the *antecedent* of the lemma and the postcondition is its *conclusion* (also known as its *proof goal*).

We state our correctness theorem as follows:

```
lemma MutualExclusion(p: Process, q: Process)
 requires Valid() && p in P && q in P
 requires cs[p] == Eating && cs[q] == Eating
 ensures p == q
{
}
```

The lemma is parameterized by two processes, p and q. It says that if the system is in a valid state (that is, the variables have values that satisfy the invariant) and if p and q are processes, each of which is in the eating state, then p and q are the same process.

The proof of a lemma is given as a method body. It follows the same rules as in Floyd/Hoare logic [7,8,12], namely that, starting from any state in which the precondition holds, every path through the body must be shown to reach the end of the body in a state where the postcondition holds. We supply a body of the lemma by a pair of curly braces, just like for the other method bodies we have seen. For the ticket-system example, it turns out that the proof can be done automatically, so I will not say more about writing proofs here.

When you supply the empty body {} of the lemma, the verifier complains that it cannot prove p == q. For this condition to follow from the lemma's antecedent, the invariant must be sufficiently strong. Our task is now to strengthen the invariant until the verifier can prove the lemma (or until we find an error in our model).

6.1 Strengthening the Invariant

The invariant is recorded in predicate Valid. So far, our invariant speaks only about P and the domains of cs and t. Writing an invariant requires a good bit of thinking and practice. Here, I will describe the additional pieces of the invariant that we need; a more comprehensive tutorial on how to write and debug invariants is out of the scope of this article.

A property we expect to hold of the ticket machinery is that the ticket dispenser may run ahead of the "serving" display, but never the other way around. We formalize this by strengthening the invariant with the condition serving <= ticket:

```
predicate Valid()
 reads this
{
 cs.Keys == t.Keys == P &&
 serving <= ticket
}
```

Having added this conjunct to Valid, Dafny immediately flags method Leave as possibly not establishing its postcondition. It frequently happens that, while strengthening an invariant, we find that showing the stronger invariant to be maintained by our operations requires further strengthenings of the invariant. So let's not be too discouraged by the fact that our strengthening increased the number of errors reported by the verifier.

The new error can help us get ideas about how to strengthen the invariant next. When `Leave(p)` increments `serving`, we expect that `ticket` is already strictly larger than `serving`—when `p` got its ticket, it would have incremented `ticket`, and ticket is never decreased by any process. This leads us to think about the property that every ticket number held by a process lies in the range from `serving` to `ticket`. Thus, we strengthen the invariant to the following condition:

```
predicate Valid()
 reads this
{
 cs.Keys == t.Keys == P &&
 serving <= ticket &&
 forall p :: p in P && cs[p] != Thinking ==> serving <= t[p] < ticket
}
```

In the new conjunct, `!=` is the operator for "not equals", `==>` is logical implication, and `<=` and `<` are the usual arithmetic comparison operators that we have chained together. The binding power of `==>` is lower than that of `&&`, as is suggested by the fact that `==>` is wider than `&&` (3 ASCII characters wide for implication versus 2 for conjunction). The relative binding powers of the other operators are like in C and Java. The type of the bound variable `p`, namely `Process`, is inferred by Dafny. (If you wonder what type is inferred, you can hover the mouse over the identifier in the Dafny IDE for Visual Studio and a tool tip will tell you this information. Of course, you can also specify the type manually, if you prefer or if the type inference were ever to fail.)

The new conjunct allows the verifier to prove the previous conjunct, but the verifier now complains that `Leave` may fail to maintain the new conjunct. The error message gives us more precise information:

```
TS.dfy(78,2): Error: A postcondition might not hold on this return path.
TS.dfy(77,12): Related location: This is the postcondition that might not hold.
TS.dfy(36,4): Related location
TS.dfy(36,56): Related location
```

It says, in the respective lines, that `Leave` may fail to establish the postcondition, that `Valid()`is the postcondition that is not established, that the `forall` is the conjunct inside `Valid()` that is not established, and that `serving <= t[p]` is the conjunct inside the quantification that is not established. Apparently, the verifier imagines a situation where some other process also has a `t` value that is equal to `serving`—if so, incrementing `serving` would violate the quantifier we just wrote. But we expect processes to have unique ticket numbers. Let's write that down by strengthening the invariant again:

```
predicate Valid()
 reads this
{
 cs.Keys == t.Keys == P &&
   serving <= ticket &&
   (forall p :: p in P && cs[p] != Thinking ==> serving <= t[p] < ticket) &&
   (forall p,q ::
       p in P && q in P && p != q && cs[p] != Thinking && cs[q] != Thinking
       ==> t[p] != t[q])
 }
```

Note the parentheses we now need around the first quantifier. I also added parentheses around the second quantifier, to make the two quantifiers cosmetically more alike.

The introduction of the new conjunct did not eliminate the complaint about the previous error. Apparently, uniqueness of ticket numbers among the processes is not enough. We also need to know that the process that is leaving the critical section is indeed one that holds a ticket number that agrees with `serving`. We update `Valid()` once more:

```
 predicate Valid()
  reads this
{
  cs.Keys == t.Keys == P &&
  serving <= ticket &&
  (forall p :: p in P && cs[p] != Thinking ==> serving <= t[p] < ticket) &&
  (forall p,q ::
     p in P && q in P && p != q && cs[p] != Thinking && cs[q] != Thinking
     ==> t[p] != t[q]) &&
  (forall p :: p in P && cs[p] == Eating ==> t[p] == serving)
}
```

This does the trick! Dafny now verifies all proof obligations: the condition we defined `Valid()` to stand for holds initially, is maintained by all atomic events, and implies mutual exclusion.

6.2 Initialization Revisited

In the constructor, we initialized both `ticket` and `serving` to `0`. This seems reasonable, but it is more specific than necessary. All we need to establish the invariant is that `ticket` and `serving` start off with the same value. We can therefore replace the second line of the constructor with

```
   ticket := serving;
```

This would set `ticket` to whatever arbitrary value `serving` happens to have on entry to the constructor. More to the point, it would make `ticket` and `serving` equal, which is enough to establish the invariant.

6.3 TLA+ Inspired Conjunctions

To improve readability, TLA+ allows conjunctions and disjunctions to be written as if they were bulleted lists. The "bullets" used are the operators for conjunction and disjunction, respectively, and the syntactic rules for grouping these pay attention to the level of indentation used in the program text. Inspired by these TLA+ expressions, Dafny allows conjunction and disjunction to be used as prefix operators instead of as infix operators. Stated differently, Dafny allows a conjunction to be preceded by `&&` and allows a disjunction to be preceded by `||`. Dafny does not pay attention to indentation, but even this simple allowance can let some formulas be formatted in a nice way. For example, we can write predicate `Valid` as follows:

```
 predicate Valid()
  reads this
{
  && cs.Keys == t.Keys == P
  && serving <= ticket
  && (forall p :: p in P && cs[p] != Thinking ==> serving <= t[p] < ticket)
  && (forall p,q ::
        p in P && q in P && p != q &&
        cs[p] != Thinking && cs[q] != Thinking
        ==> t[p] != t[q])
  && (forall p :: p in P && cs[p] == Eating ==> t[p] == serving)
}
```

Note, the prefix allowance does not change the binding power of operators. This means that parentheses are still needed as before. In particular, note in this example that the `forall` quantifiers need parentheses—they do not end at the end of the line as the bullet syntax may lead you to believe. Another important operator to consider is implication, which binds more loosely than conjunction. Thus, if `Valid` contained a conjunct that was an implication, then it would need parentheses.

7 Event Scheduling

In what I have shown so far, the constructor and methods of class `TicketSystem` are never called. In other words, in the program text we have written, we have defined some atomic events, but we have not defined any scheduler that invokes the events and produces a particular interleaving. Instead, we have left it to our imagination that the events can be invoked at any time. This is typical. Models usually include only the initialization condition and the atomic events. The scheduler is tacit, and it is understood that the events can occur at any time, as long as the precondition that is guarding their execution holds.

For the tutorial purpose of seeing what this implicit scheduler that we are imagining looks like, let us write a scheduler explicitly. The scheduler will exhibit a high degree of nondeterminism, so we won't actually restrict the possible interleavings. The scheduler will still serve two purposes. One purpose is to demonstrate that it is always possible to schedule a process. If we accidentally had left out some event (say, method `Leave`), then our scheduler would not know what to do with a process in the `Eating` state. The second purpose is for us to warm up to issues of "fair" scheduling, which I will address in Sect. 8.1.

7.1 A Scheduler

We write a method `Run` that takes a nonempty set of processes and repeatedly calls these to perform an atomic event. The method looks like this:

```
 method Run(processes: set<Process>)
  requires processes != {}
  decreases *
{
  var ts := new TicketSystem(processes);
  while true
    invariant ts.Valid()
    decreases *
  {
    var p :| p in ts.P;
    match ts.cs[p] {
      case Thinking => ts.Request(p);
      case Hungry => ts.Enter(p);
      case Eating => ts.Leave(p);
    }
  }
}
```

Here is an explanation of this code.

– Because the loop never terminates, it must be marked with `decreases *`. This allows the loop to go on forever and it causes the verifier to omit termination checks. Similarly, a method that contains a nonterminating loop must itself be declared with `decreases *`.
– Reasoning about loops is done via *loop invariants* [7,8,12]. These are conditions that hold at the very top of every loop iteration. Loop invariants simplify the problem of reasoning about all possible iteration traces of the loop to reasoning about just one, arbitrary iteration. The use of loop invariants bears resemblance to the use of an inductive hypothesis in mathematics. In this example, the `invariant` declaration says that the invariant of the ticket system is maintained by the loop.
– The assignment statement `x :| R` sets variable `x` to a value satisfying the boolean expression `R`. The statement gives rise to a proof obligation that such an `x` exists. As we have written our program so far, the verifier complains that this proof obligation does not hold. That is because there is no connection between `processes` (which is known to be nonempty) and `ts.P`. To correct this problem, we add the postcondition

```
    ensures P == processes
```

  to the `TicketSystem` constructor. The `:|` assign-such-that statement in the `Run` method is now proved to be legal. Its effect is to introduce a variable `p` to stand for an arbitrary process from the set `ts.P`.
– The `match` statement continues execution with one of the given cases, depending on the value of `ts.cs[p]`. The statement gives rise to a proof obligation that there is a case for every possible value of the given expression. This is where the verifier would complain if we had forgotten to define method `Leave` and left out the case for `Eating`. (Try commenting this line out and you will see.)

Notice that it is the scheduler—that is, our Run method—that picks the process p to schedule next and controls which of the three events to invoke for process p. We have written the Run method so that p is picked arbitrarily among the processes, but having picked p, the choice of which event to invoke is determined by the control state of that process, cs[p]. This corresponds to what a scheduler really does: How a scheduler picks which process to schedule next may be more sophisticated than the arbitrary choice in our Run method, but once it has made that decision, the scheduler must be sure to start the process in (that is, set the program counter of the process to) the control state where the process last left off.

7.2 Guards of Events

Event Enter in our example models what in Sect. 4 I denoted by the pseudo-code statement **wait until**. You may think of the way we wrote method Enter as corresponding to a "busy waiting" (aka "spinning") implementation of **wait until**. If the scheduler picks a process in the Hungry state when its t value is not equal to serving, then that process just "skips" (see the if statement in method Enter). This is a common design for modeling a process that is suspended on a condition. Let me explain two alternative designs.

One alternative design is to replace our Enter method above with the following two methods:

```
method Enter0(p: Process)
 requires Valid() && p in P && cs[p] == Hungry && t[p] == serving
 modifies this
 ensures Valid()
{
 cs := cs[p := Eating];
}

method Enter1(p: Process)
 requires Valid() && p in P && cs[p] == Hungry && t[p] != serving
 modifies this
 ensures Valid()
{
}
```

Method Enter0 corresponds to the then branch of the if statement in Enter and method Enter1 corresponds to the else branch. Note that the guard of the if statement in Enter has become part of the preconditions of Enter0 and (in negated form) Enter1. By breaking Enter into two methods in this design of the model, we are essentially thinking of transitioning from Hungry to Eating as one event (Enter0) and remaining hungry as another event (Enter1). This design is also common.

To see how the alternative design makes sense, it is instructive to look at how it impacts the scheduler. This is especially important since, as I have mentioned, the scheduler is usually left tacit. To deal with Enter0 and Enter1, we change the Hungry case of method Run to:

```
case Hungry =>
   if ts.t[p] == ts.serving {
     ts.Enter0(p);
   } else {
     ts.Enter1(p);
   }
```

A second alternative design is to use only `Enter0`, not `Enter1`. In the original design, we thought of `Enter` as busy waiting—the process is scheduled, inspects `t` and `serving`, and may then find that it has nothing to do but wait further. By including only `Enter0`, our model reflects the thinking that a process in the `Hungry` state gets scheduled only when it can proceed. In other words, in this design, the scheduler understands and directly supports the **`wait until`** operation.

To accommodate this second alternative design, one possible way to rewrite the scheduler is to drop the else branch in the case for `Hungry`. If the scheduler happens to pick `p` to be a hungry process that is not yet being served, then the scheduler skips. In other words, an iteration of the scheduler loop that picks such a `p` ends up being an unproductive iteration.

We can make all of the scheduler's loop iterations productive if we make the selection of `p` more precise. We do that by strengthening the condition to this:

```
var p :| p in ts.P && (ts.cs[p] == Hungry ==> ts.t[p] == ts.serving);
```

With this condition, the verifier complains that it cannot always prove the existence of such a `p`. This is a valid complaint, because if the ticket system were coded incorrectly, we could end up in a situation where the scheduler cannot schedule any process. Such a situation reflects a deadlock.

We may be tempted to prove that a deadlock cannot occur and that the scheduler always has some process to schedule. However, I argue that the responsibility of avoiding deadlocks does not lie with the scheduler. Instead, if we want to model a scheduler that directly supports an **`wait until`** operation, then the scheduler is allowed to terminate if the processes get themselves into a deadlock situation. Thus, the scheduler in this second alternative design looks like:

```
var ts := new TicketSystem(processes);
while exists p :: p in ts.P &&
             (ts.cs[p] == Hungry ==> ts.t[p] == ts.serving)
   invariant ts.Valid()
   decreases *
 {
   var p :| p in ts.P && (ts.cs[p] == Hungry ==> ts.t[p] == ts.serving);
   match ts.cs[p] {
     case Thinking => ts.Request(p);
     case Hungry => ts.Enter0(p);
     case Eating => ts.Leave(p);
   }
 }
```

Note that the loop guard no longer uses the condition `true` and note that the `Hungry` case only calls `Enter0`.

7.3 Recording or Following a Schedule

Before moving on to a different representation of the model where we can prove that the ticket system does not exhibit deadlocks, let us consider one small change to the `Run` method. It is to add some code that records the scheduling choices that `Run` makes. We can easily do this by instrumenting `Run` with

```
var schedule := [];
```

before the loop and

```
schedule := schedule + [p];
```

inside the loop. Similarly, if we wish, we can also record the sequence of states, here as a sequence of 4-tuples, by instrumenting `Run` with

```
var trace := [(ts.ticket, ts.serving, ts.cs, ts.t)];
```

before the loop and

```
trace := trace + [(ts.ticket, ts.serving, ts.cs, ts.t)];
```

at the end of the loop.

Instead of recording the decisions that `Run` makes, we can change `Run` to free it from making any decisions at all. We accomplish this by letting `Run` take a predetermined schedule as a parameter. This schedule has the form of a function from times to processes.

```
 method RunFromSchedule(processes: set<Process>, schedule: nat -> Process)
  requires processes != {}
  requires forall n :: schedule(n) in processes
  decreases *
{
  var ts := new TicketSystem(processes);
  var n := 0;
  while true
    // ...
  {
    var p := schedule(n);
    match  // ...
    n := n + 1;
  }
}
```

The type `nat -> Process` denotes total functions from natural numbers to processes.

8 Liveness Properties

So far, I have covered the basics of modeling concurrency by breaking down a system into atomic events. Dafny's methods give a convenient representation of each atomic event, and I have shown how the system invariant can be captured in a predicate `Valid()` that is a pre- and postcondition of every method. To prove that the system can only reach "safe" states (for example, at most one philosopher is in the kitchen at any one time), we prove that the system invariant implies the safety property of interest.

There is another class of properties that you may want to prove about a concurrent system: liveness properties. Whereas a safety property says that the system remains in safe states, a liveness property says that the system eventually leaves anxious states. For example, in our ticket system, a desired liveness property is that a hungry philosopher eventually eats. In other words, this liveness property says that the system does not forever remain in a set of states where a particular philosopher is hungry; the system eventually leaves that set of "anxious" states.

8.1 Fair Scheduling

To prove any liveness property, we need to know more about the scheduler. For example, suppose a philosopher A enters the hungry state and that the scheduler, from that time onwards, decides to pick another philosopher B exclusively. Philosopher A will then remain hungry forever and our desired liveness property does not hold. In fact, in our ticket system, such a schedule will also eventually cause B to get stuck in the hungry state, skipping each time the scheduler gives it control.

When we reason about liveness properties, we work under the assumption that the scheduler is *fair*, meaning that it cannot ignore some process forever. In contrast, no fairness assumption is needed when reasoning about safety properties, since restricting our attention to fair schedules does not change the set of possibly reachable states.

8.2 State Tuple

When we were proving safety properties, it was convenient to formalize our model using methods. This let us make use of the program statements in Dafny. To prove liveness properties, we frequently have a need to refer to states that will eventually happen in the future. This is more conveniently done using a different formalization. So, we are going to start afresh.

Processes and control state are as before:

```
type Process(==)
const P: set<Process>
datatype CState = Thinking | Hungry | Eating
```

Instead of modeling the state using the fields of a class, we will model the state as a tuple. We define the tuple as an immutable record:

```
datatype TSState = TSState(ticket: int,
                           serving: int,
                                    cs: map<Process, CState>,
                                    t: map<Process, int>)
```

The system invariant is like before, but speaks about the fields of a `TSState` record rather than the fields of an object. Not using objects, there are no issues of framing (like `reads` clauses) to worry about.

```
predicate Valid(s: TSState)
{
 s.cs.Keys == s.t.Keys == P &&
 s.serving <= s.ticket &&
 (forall p :: p in P && s.cs[p] != Thinking ==>
          s.serving <= s.t[p] < s.ticket) &&
 (forall p,q ::
    p in P && q in P && p != q &&
    s.cs[p] != Thinking && s.cs[q] != Thinking
    ==> s.t[p] != s.t[q]) &&
 (forall p :: p in P && s.cs[p] == Eating ==> s.t[p] == s.serving)
}
```

In Sect. 6.1, we did the hard work of strengthening the invariant to prove the mutual-exclusion safety property. This lets us simply restate the automatically proved theorem here.

```
lemma MutualExclusion(s: TSState, p: Process, q: Process)
  requires Valid(s) && p in P && q in P
  requires s.cs[p] == Eating && s.cs[q] == Eating
  ensures p == q
{
}
```

8.3 State Transitions

Following the format in TLA+, we define our model by a predicate `Init` that describes the possible initial states and a predicate `Next` that describes the possible transitions from one state to the next.

```
predicate Init(s: TSState)
{
  s.cs.Keys == s.t.Keys == P &&
  s.ticket == s.serving &&
  forall p :: p in P ==> s.cs[p] == Thinking
}

predicate Next(s: TSState, s': TSState)
{
  Valid(s) &&
  exists p :: p in P && NextP(s, p, s')
}
```

We define `Next` to hold only for pairs of states where the first state satisfies the system invariant, `Valid`, but we do not mention either `Valid(s)` in `Init` or

Valid(s') in Next. It is better to prove the invariance of Valid separately. Once we have defined NextP, you can do that by a lemma like:

```
  lemma Invariance(s: TSState, s': TSState)
   ensures Init(s) ==> Valid(s)
   ensures Valid(s) && Next(s, s') ==> Valid(s')
```

We define predicate NextP(s, p, s') to say that a process p may take an atomic step from state s to state s'. As we have seen before, such an atomic step is Request, Enter, or Leave.

```
 predicate NextP(s: TSState, p: Process, s': TSState)
  requires Valid(s) && p in P
{
  Request(s, p, s') || Enter(s, p, s') || Leave(s, p, s')
}
```

Predicate NextP is defined to have a precondition, which means it can be used only when its parameters satisfy the required condition. This is fitting for our purposes, since the call to NextP in Next is guarded by Valid(s) and p in P, and it means we can assume these conditions (rather than repeat them) in the body of NextP.

Finally, we write the predicates for the three atomic events:

```
 predicate Request(s: TSState, p: Process, s': TSState)
   requires Valid(s) && p in P
{
   s.cs[p] == Thinking &&
   s'.ticket == s.ticket + 1 &&
   s'.serving == s.serving &&
   s'.t == s.t[p := s.ticket] &&
   s'.cs == s.cs[p := Hungry]
}

predicate Enter(s: TSState, p: Process, s': TSState)
   requires Valid(s) && p in P
{
   s.cs[p] == Hungry &&
   s'.ticket == s.ticket &&
   s'.serving == s.serving &&
   s'.t == s.t &&
   ((s.t[p] == s.serving && s'.cs == s.cs[p := Eating]) ||
    (s.t[p] != s.serving && s'.cs == s.cs))
}

predicate Leave(s: TSState, p: Process, s': TSState)
  requires Valid(s) && p in P
{
  s.cs[p] == Eating &&
  s'.ticket == s.ticket &&
  s'.serving == s.serving + 1 &&
  s'.t == s.t &&
  s'.cs == s.cs[p := Thinking]
}
```

8.4 Schedules and Traces

As we got a glimpse of in Sect. 7.3, a schedule is a function from times (represented by natural numbers) to processes.

```
type Schedule = nat -> Process

predicate IsSchedule(schedule: Schedule)
{
  forall i :: schedule(i) in P
}
```

A trace is a function from times to ticket-system states. Such a trace is possible for a given schedule if the trace starts in a state satisfying `Init` and every pair of consecutive states in the trace is allowed as an atomic event by the process scheduled at that time.

```
type Trace = nat -> TSState

predicate IsTrace(trace: Trace, schedule: Schedule)
  requires IsSchedule(schedule)
{
  Init(trace(0)) &&
  forall i: nat ::
    Valid(trace(i)) && NextP(trace(i), schedule(i), trace(i+1))
}
```

Finally, a schedule is fair if every process occurs infinitely often. That is, for any process `p`, no matter how many steps you have already taken—say, `n` steps—there is still a next time, `n'`, where `p` will be scheduled.

```
predicate FairSchedule(schedule: Schedule)
{
  IsSchedule(schedule) &&
  forall p,n :: p in P ==> HasNext(schedule, p, n)
}
predicate HasNext(schedule: Schedule, p: Process, n: nat)
{
  exists n' :: n <= n' && schedule(n') == p
}
```

8.5 Currently Served Process

Leading up to proving the desired liveness property of our ticket system, I define two new ingredients. The first new ingredient is a function that tells us which process is being served, that is, which process is holding the ticket number shown on the "serving" display. Not every ticket-system state has a currently served process, but we expect that if *some* process is hungry or eating, then there is a currently served process. As it turns out, we will need the currently served process only when we know some process is hungry, so we attempt to define:

```
function CurrentlyServedProcess(s: TSState): Process
 requires Valid(s) && exists p :: p in P && s.cs[p] == Hungry
{
 var q :| q in P && s.cs[q] != Thinking && s.t[q] == s.serving;
 q
}
```

However, as we write this definition, the verifier complains that it cannot prove there is such a `q` (remember that the `:|` operator gives rise to a proof obligation that there exists such a `q`). The reason is that our system invariant is not strong enough. We need to strengthen it further to say that every ticket number from `s.serving` to `s.ticket` is being used.

We define a predicate

```
predicate TicketIsInUse(s: TSState, i: int)
 requires s.cs.Keys == s.t.Keys == P
{
 exists p :: p in P && s.cs[p] != Thinking && s.t[p] == i
}
```

and use it in the definition of `Valid`:

```
predicate Valid(s: TSState)
{
 ... &&
 (forall i :: s.serving <= i < s.ticket ==> TicketIsInUse(s, i))
}
```

Now, there does exist a `q` like we want in `CurrentlyServedProcess`, but the verifier needs help to do the proof. Adding an `assert` in the body of the function is enough of a hint for the verifier:

```
function CurrentlyServedProcess(s: TSState): Process
 requires Valid(s) && exists p :: p in P && s.cs[p] == Hungry
{
 assert TicketIsInUse(s, s.serving);
 var q :| q in P && s.cs[q] != Thinking && s.t[q] == s.serving;
 q
}
```

If you did the exercise of including lemma `Invariance`, you will find that the strengthened invariant causes the automatic proof of `Invariance` no longer to go through. Like `CurrentlyServedProcess`, it also needs a hint, but I will not go through the details here.

8.6 Next Step of a Process

Fairness tells us there exists some future time in a schedule where a process `p` is scheduled. Given `HasNext(schedule, p, n)`, we can use a statement

```
var u :| n <= u && schedule(u) == p;
```

to obtain such a time, u. The second new ingredient we need for our liveness proof is a way to find a future time, n', where certain properties hold. To obtain n', we can iterate up from n, since u provides us with an upper bound for the iteration.

In our application, p is the currently served process, n is the current time, and n' is the next time that p is scheduled. We prove that serving is unchanged from time n to time n', that p remains in the same control state and holds the same ticket in n as in n', and that all hungry processes in n are still hungry in n' and hold the same ticket in n' as in n.

Brace yourself for two surprises. One surprise is that we can formulate this ingredient as a lemma that has an out-parameter! We usually think of a lemma as establishing some condition. In mathematics, it is common that a lemma would establish the existence of an n' with certain properties. In Dafny, a lemma is simply a method that isn't compiled into code, and therefore it is just as natural for a lemma to have out-parameters as it is for the lemma to have in-parameters. With an out-parameter, we might as well return the n' whose existence the lemma has established. The second surprise is that the proof uses a loop! In mathematics, lemmas tend to be recursive—that is, a lemma calls itself to obtain what is known as the induction hypothesis. In Dafny, where a lemma is just a method, the body of the lemma can use iteration as well as recursion. In this case, I find it natural to express the proof by iteration, since our strategy is to iterate up from n toward u until we find the n' we are looking for.

Here is the lemma:

```
 lemma GetNextStep(trace: Trace, schedule: Schedule, p: Process, n: nat) returns (n': nat)
  requires FairSchedule(schedule) && IsTrace(trace, schedule) && p in P
  requires trace(n).cs[p] != Thinking && trace(n).t[p] == trace(n).serving
  ensures n <= n' && schedule(n') == p
  ensures trace(n').serving == trace(n).serving
  ensures trace(n').cs[p] == trace(n).cs[p]
  ensures trace(n').t[p] == trace(n).t[p]
  ensures forall q :: q in P && trace(n).cs[q] == Hungry ==>
            trace(n').cs[q] == Hungry && trace(n').t[q] == trace(n).t[q]
{
  assert HasNext(schedule, p, n);
  var u :| n <= u && schedule(u) == p;
  n' := n;
  while schedule(n') != p
    invariant n' <= u
    invariant trace(n').serving == trace(n).serving
    invariant trace(n').cs[p] == trace(n).cs[p]
    invariant trace(n').t[p] == trace(n).t[p]
    invariant forall q :: q in P && trace(n).cs[q] == Hungry ==>
                trace(n').cs[q] == Hungry && trace(n').t[q] == trace(n).t[q]
    decreases u - n'
  {
    n' := n' + 1;
  }
}
```

The loop invariant is the same as the postcondition of the lemma, except for the postcondition schedule(n') == p, which is obtained as the negation of the loop guard. The decreases clause is used to prove termination of the loop.

It gives a natural-number valued expression whose value decreases with every iteration [8].

8.7 Liveness Theorem

Finally, we are ready to state and prove the liveness theorem. It states that a hungry process eventually eats. Instead of just showing the existence of a future time when the process eats, the lemma returns that time, similarly to what we saw for the `GetNextStep` lemma. And as in the proof of `GetNextStep`, I find that the proof of the liveness theorem is naturally formulated as a loop. If you think like a programmer, this liveness-theorem proof is just an algorithm that finds (and returns) the next time the hungry process eats.

```
lemma Liveness(trace: Trace, schedule: Schedule, p: Process, n: nat) returns (n': nat)
  requires FairSchedule(schedule) && IsTrace(trace, schedule) && p in P
  requires trace(n).cs[p] == Hungry
  ensures n <= n' && trace(n').cs[p] == Eating
{
  n' := n;
  while true
    invariant n <= n' && trace(n').cs[p] == Hungry
    decreases trace(n').t[p] - trace(n').serving
  {
    // find the currently served process and follow it out of the kitchen
    var q := CurrentlyServedProcess(trace(n'));
    if trace(n').cs[q] == Hungry {
      n' := GetNextStep(trace, schedule, q, n');
      n' := n' + 1;  // take the step from Hungry to Eating
      if p == q {
        return;
      }
    }
    n' := GetNextStep(trace, schedule, q, n');
    n' := n' + 1;  // take the step from Eating to Thinking
  }
}
```

9 Conclusion

A central idea in reasoning about concurrency is to break up the behavior of each process into atomic events whose execution may be interleaved with the atomic events of other processes. I have shown how to go from the pseudo code of a simple mutual-exclusion protocol to two formalizations of atomic events. The atomic events can be verified to maintain some system invariant, and safety properties, like mutual exclusion, are proved as logical consequences of the system invariant. In the second formalization, I also expressed and proved a liveness property. The key here is to be able to reason about events that will eventually take place in the execution of the program. The liveness proofs I showed bear some resemblance to search algorithms.

When using Dafny to model concurrency, the Dafny language and verifier provide a logical foundation. The fact that Dafny is a programming language that can be compiled is not the central point. Indeed, we would not be interested in compiling and running the code that we wrote as part of our models. Nevertheless, the notation that Dafny borrows from other programming languages lowers the bar for entry into new formalization projects. The automation provided by the Dafny verifier and the rapid verification feedback provided in the Dafny IDEs also aid in the Dafny usage experience. For example, in the safety-property theorem, we only needed to supply the invariant and then all the proofs were carried out automatically.

Acknowledgments. Andreas Podelski brought up the problem of this ticket system at meeting 54 (Saint Petersburg, Russia, June 2013) of the IFIP Working Group 2.3, where several formalizations were subsequently presented. Xinhaoyuan on github proposed the prefix allowance of conjunctions and disjunctions in Dafny. I thank Shuo Chen for comments on an earlier draft of this article.

References

1. Abrial, J.-R.: Modeling in Event-B: System and Software Engineering. Cambridge University Press, Cambridge (2010)
2. Abrial, J.-R.: Mini-course around Event-B and Rodin, June 2011. https://www.microsoft.com/en-us/research/video/mini-course-around-event-b-and-rodin/
3. Abrial, J.-R., Butler, M., Hallerstede, S., Hoang, T.S., Mehta, F., Voisin, L.: Rodin: an open toolset for modelling and reasoning in Event-B. Softw. Tools Technol. Transf. **12**(6), 447–466 (2010)
4. Back, R.-J., Sere, K.: Action systems with synchronous communication. In: Olderog, E.-R. (ed.) Proceedings of the IFIP TC2/WG2.1/WG2.2/WG2.3 Working Conference on Programming Concepts, Methods and Calculi (PROCOMET 1994). IFIP Transactions, vol. A-56, pp. 107–126. North-Holland, June 1994
5. Chandy, K.M., Misra, J.: Parallel Program Design: A Foundation. Addison-Wesley, Boston (1988)
6. Dafny online (2017). http://rise4fun.com/dafny
7. Floyd, R.W.: Assigning meanings to programs. In: Proceedings of the Symposium on Applied Mathematics, vol. 19, pp. 19–32. American Mathematical Society (1967)
8. Gries, D.: The Science of Programming. MCS. Springer-Verlag, New York (1981). https://doi.org/10.1007/978-1-4612-5983-1
9. Hatcliff, J., Leavens, G.T., Leino, K.R.M., Müller, P., Parkinson, M.: Behavioral interface specification languages. ACM Comput. Surv. **44**(3), 16:1–16:58 (2012)
10. Hawblitzel, C., et al.: IronFleet: proving practical distributed systems correct. In: Miller, E.L., Hand, S. (eds.) Proceedings of the 25th Symposium on Operating Systems Principles, SOSP 2015, pp. 1–17. ACM, October 2015
11. Herbert, L., Leino, K.R.M., Quaresma, J.: Using Dafny, an automatic program verifier. In: Meyer, B., Nordio, M. (eds.) LASER 2011. LNCS, vol. 7682, pp. 156–181. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-35746-6_6
12. Hoare, C.A.R.: An axiomatic basis for computer programming. Commun. ACM **12**(10), 576–583 (1969)
13. Jackson, D.: Software Abstractions: Logic, Language, and Analysis. MIT Press, Cambridge (2006)
14. Koenig, J., Leino, K.R.M.: Getting started with Dafny: a guide. In: Nipkow, T., Grumberg, O., Hauptmann, B. (eds.) Software Safety and Security: Tools for Analysis and Verification. NATO Science for Peace and Security Series D: Information and Communication Security, vol. 33, pp. 152–181. IOS Press (2012). Summer School Marktoberdorf 2011 lecture notes

15. Lamport, L. (ed.): Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers. Addison-Wesley Professional, Boston (2002)
16. Lamport, L.: The TLA+ video course, March 2017. http://lamport.azurewebsites.net/video/videos.html
17. Leino, K.R.M.: Specification and verification of object-oriented software. In: Broy, M., Sitou, W., Hoare, T. (eds.) Engineering Methods and Tools for Software Safety and Security. NATO Science for Peace and Security Series D: Information and Communication Security, vol. 22, pp. 231–266. IOS Press (2009). Summer School Marktoberdorf 2008 lecture notes
18. Leino, K.R.M.: Dafny: an automatic program verifier for functional correctness. In: Clarke, E.M., Voronkov, A. (eds.) LPAR 2010. LNCS (LNAI), vol. 6355, pp. 348–370. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-17511-4_20
19. Leino, K.R.M.: Developing verified programs with Dafny. In: Notkin, D., Cheng, B.H.C., Pohl, K. (eds.) 35th International Conference on Software Engineering, ICSE 2013, pp. 1488–1490. IEEE Computer Society (2013)
20. Leino, K.R.M.: Accessible software verification with Dafny. IEEE Software **34**(6), 94–97 (2017)