Concurrency and Transactional Memory in Dafny

Elisa Guerrant, Clément Burgelin, Hugo Hueber
{*elisa.guerrant, clement.burgelin, hugo.hueber*}*@epfl.ch*

***Abstract*—Our project consists of two main parts: to model concurrency with the Dafny programming language and prove several properties about the program using the available verifier tools, such as safety and liveness, and to realise a model of a lock-based transactional memory (TM). We first understood and mastered the Dafny programming language, using the guide from [1] and by reproducing [2]. Secondly, we made a simple lock-based TM with invisible-reads in this language, using an algorithm similar to one found on [3], and tried proving properties on it. Finally, we rewrote the TM system in stainless to compare the languages.**

Our implementations of the different parts is available at : https://github.com/Zyfarok/fv2020-concurrency-and-tms

***Keywords*-Dafny language, concurrency, transactional memory, Stainless, Scala**

I. Introduction

Formal verification of concurrency is not a trivial task. Most formal verification languages and programs do not support concurrency natively, and therefore it must be simulated in order to make verification realisable ([4]). We based our project on two papers, "Modeling Concurrency in Dafny" ([2]) and "The Semantics of Progress in Lock-Based Transactional Memory" ([3]).

The first paper shows how to simulate concurrency in the Dafny language, allowing one to do proofs on concurrent systems. It deals with a concrete example, a ticket system, where only one person/process can access a resource at the same time. The second paper describes several transactional memory algorithms and provides direction into how to prove "strong progressiveness" (a liveness property) of those. A transaction memory algorithm aims to allow a group of read and write instructions to execute in an atomic way by controlling access to a shared memory in a concurrent system.

A. Code

The code written for this project is publicly available [5].

II. Dafny overview

Dafny is a language and program verifier which checks algorithms for functional correctness. According to the specifications, Dafny is imperative, sequential, supports generic classes, dynamic allocation, and various other elements specific to imperative languages.

Dafny borrows its structure from C#, but has many features of its own. This section will go over some interesting or subtle elements specific to the language:

Figure 1. Dafny logo.

Pre- and post-conditions: Dafny uses a special structure to indicate pre-conditions (`requires`) and post-conditions (`ensures`). These conditions need to be made explicit when writing methods and functions.

Functions and methods: Dafny supports functions and methods. Methods are used in most programming languages, with bodies that may consist of multiple expressions, while function bodies must consist of exactly one expression with the correct type. Functions are useful to prove statements and be a quick, mathematical reference, rather than have a tedious method that must be formally proven each time.

Loops: A loop invariant is a logical assertion related to a program loop that must evaluate to true before each iteration of the loop. It is difficult to use because of the need to preserve this property while also making changes to the program state within the body of the loop. In Dafny, the keyword `invariant` is used along with a Boolean proposition and is extensively used to prove a lot of properties.

One particularity of the Dafny language is that it needs help to prove a property for some cases within loops and recursion: we need to explicitly tell Dafny what to do with the variables and when to finish. For that, the termination is made explicit with the `decreases` keyword, for instance `decreases n - i` with invariant `0 <= i <= n`.

Reading frame: Basically, a reading frame is a "read only" flag on a variable/array predicate. Predicates cannot read the object without it. It is used so that we do not access things that should not be accessed. It additionally specifies a set of memory locations that the function is allowed to access. For methods, we can specify what we are modifying with the 'modifies' keyword.

Quantifiers, predicates, logical proofs: Dafny also allows the use of quantifiers and predicates in code, much in the same way that they could be used in logical proofs. In particular, quantifiers work exactly as in formal proofs, with the keywords `forall` or `exists`. "Bound variables" are temporary variable names for each element of a set being explored.

Consider `assert forall k :: 0 <= k < a.Length ==> a[k] != k;`

The structure of the statement is as follows: `k` is the bound variable, followed by `::`, `0` is the quantified property of type `bool`, followed by `==>`, and finally the corresponding property.

Dafny has also the `predicate` keyword, which can be summarized as a function that returns a bool. It can be used to write shorter code and can use quantifiers.

Short-circuiting: Dafny also has one more feature that is of particular interest to us: short-circuiting ([6]). When doing proofs, the right-hand side argument of a boolean operation will only be evaluated if the first argument cannot be used to determine the overall value of the expression (eg. in the expression false AND true, the second argument is not evaluated). This can be used to speed up some proofs and computations.

III. Modeling Concurrency in Dafny

Dafny is a good language to work on imperative code and simulating concurrency. We wanted to work on this language to extend our possibilities and knowledge of formal verification.

The first paper we used in our project deals with two models of a ticket system, each showing a different possible implementation and proving different specific properties of concurrency, namely safety (models 1 and 2) and liveness (only model 2).

A. Ticket system reminder

The ticket system is directly inspired by the dining philosophers problem and deals with a group of philosophers at work. The philosophers are occasionally hungry, and wish to have access to a kitchen, which can only hold one person at a time. A ticket system is used to determine who can access the kitchen at any given moment.

Philosophers are equivalent to processes. Each process has three states, namely Thinking, Hungry and Eating. When the process would like to access the kitchen (shared resource), they receive a ticket with a unique number and

Figure 2. Our philosophers in a situation that should not happen with a correct algorithm.

enter the Hungry state. When the "now serving" number corresponds to the number on their ticket, they can access the kitchen (Eating). The "now serving" number increases when a process stops using the protected space (returns to Thinking), allowing subsequent processes to access the resource.

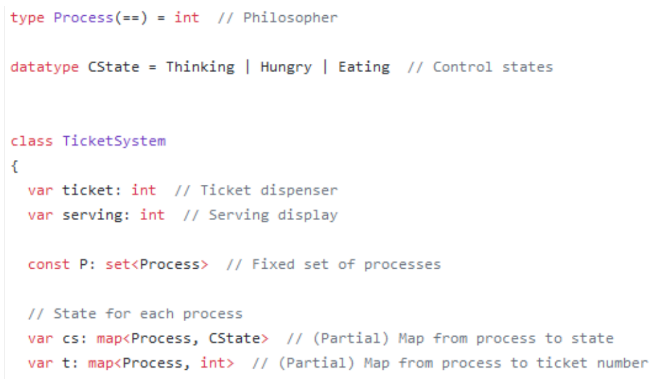
B. Safety implementation

Safety is the idea that nothing bad will ever happen, essentially the property that states that a thread (or process in our case) is able to function correctly when it is executed simultaneously within the same address space by several other threads. In Dafny, we prove this by showing that certain invariants are true at all points during the execution of the algorithm. (Technically, since Dafny operates using pre- and post-conditions, we can only show that properties hold true before and after the function body is executed. However, since the program is structured so that each atomic action taken by the processes is in a separate function, this is equivalent to stating that the invariants hold true at all points during program execution.)

1) Model: The first model only implements safety, not liveness. It is a rather straightforward model, an "ecosystem" made under a class, as seen on Fig. 3, with a constructor, a global map of process to state, a global counter for tickets, and a global counter for the "now serving" parameter. The previously discussed atomic events are modeled with functions.

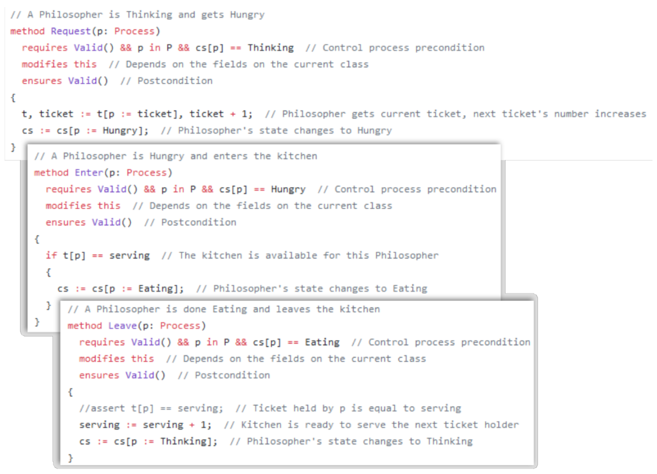
2) Atomic events: As previously stated, our states are "Thinking", "Hungry", and "Eating" (Fig. 4). Each function call that represent an atomic action checks that the state of the processes is valid before and after executing the body of the function.

3) `Valid` predicate: The `Valid` predicate (Fig. 5), which is used throughout the model, is a way to check that each atomic event is, indeed, valid. It defines whether a Process can do a specific action and is used both as a pre-condition and a post-condition.



```
type Process(==) = int  // Philosopher

datatype CState = Thinking | Hungry | Eating  // Control states


class TicketSystem
{
  var ticket: int  // Ticket dispenser
  var serving: int  // Serving display

  const P: set<Process>  // Fixed set of processes

  // State for each process
  var cs: map<Process, CState>  // (Partial) Map from process to state
  var t: map<Process, int>  // (Partial) Map from process to ticket number
```

Figure 3. Type and begining of the class.

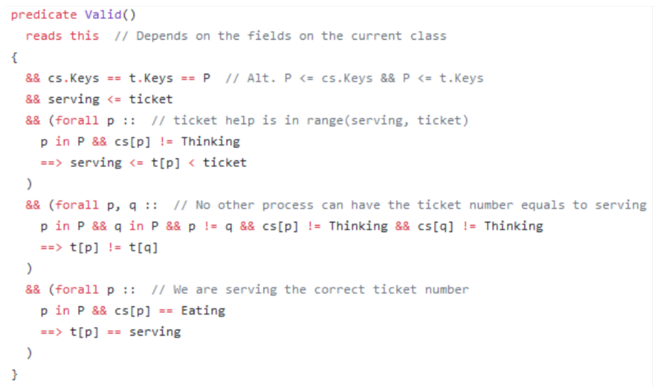


```
// A Philosopher is Thinking and gets Hungry
method Request(p: Process)
  requires Valid() && p in P && cs[p] == Thinking  // Control process precondition
  modifies this  // Depends on the fields on the current class
  ensures Valid()  // Postcondition
{
  t, ticket := t[p := ticket], ticket + 1;  // Philosopher gets current ticket, next ticket's number increases
  cs := cs[p := Hungry];  // Philosopher's state changes to Hungry
}
```

```
// A Philosopher is Hungry and enters the kitchen
method Enter(p: Process)
  requires Valid() && p in P && cs[p] == Hungry  // Control process precondition
  modifies this  // Depends on the fields on the current class
  ensures Valid()  // Postcondition
{
  if t[p] == serving  // The kitchen is available for this Philosopher
  {
    cs := cs[p := Eating];  // Philosopher's state changes to Eating
  }
}
```

```
// A Philosopher is done Eating and leaves the kitchen
method Leave(p: Process)
  requires Valid() && p in P && cs[p] == Eating  // Control process precondition
  modifies this  // Depends on the fields on the current class
  ensures Valid()  // Postcondition
{
  //assert t[p] == serving;  // Ticket held by p is equal to serving
  serving := serving + 1;  // Kitchen is ready to serve the next ticket holder
  cs := cs[p := Thinking];  // Philosopher's state changes to Thinking
}
```

Figure 4. Atomic states modeled as functions.



```
predicate Valid()
  reads this  // Depends on the fields on the current class
{
  && cs.Keys == t.Keys == P  // Alt. P <= cs.Keys && P <= t.Keys
  && serving <= ticket
  && (forall p ::  // ticket help is in range(serving, ticket)
    p in P && cs[p] != Thinking
    ==> serving <= t[p] < ticket
  )
  && (forall p, q ::  // No other process can have the ticket number equals to serving
    p in P && q in P && p != q && cs[p] != Thinking && cs[q] != Thinking
    ==> t[p] != t[q]
  )
  && (forall p ::  // We are serving the correct ticket number
    p in P && cs[p] == Eating
    ==> t[p] == serving
  )
}
```

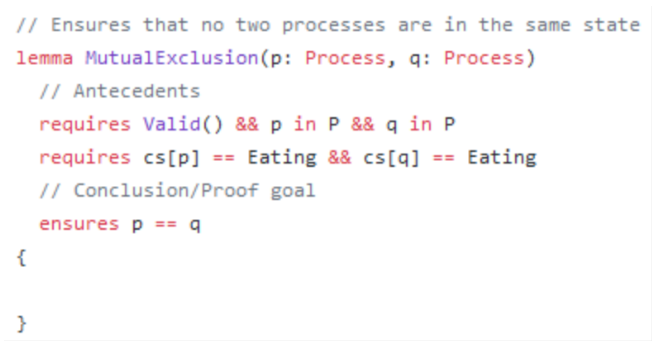
Figure 5. The `Valid` predicate.

This predicate will check the following information for each atomic event:

1) Each process is part of the process list.
2) The serving number is less or equal than the number of the last ticket to be issued.
3) Each ticket is in the range of the available tickets.
4) No other process has the number of the currently served ticket.
5) The correct ticket number is being served.

With all these properties holding, we can finally check for the final condition for our kitchen not to be overwhelmed.

4) The main lemma: `MutualExclusion`: The main thing that needs to be enforced with this algorithm is that only one process can access the shared resource at a time. The Lemma described here (Fig. 6) will be used to check that two different Process cannot be 'Eating' at the same time, that is, that the shared resource is only accessed by one process at a time. 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.



```
// Ensures that no two processes are in the same state
lemma MutualExclusion(p: Process, q: Process)
  // Antecedents
  requires Valid() && p in P && q in P
  requires cs[p] == Eating && cs[q] == Eating
  // Conclusion/Proof goal
  ensures p == q
{


}
```

Figure 6. Mutual exclusion, the key to concurrency.

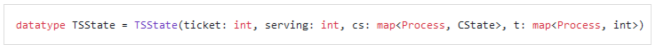
Our Ticket System is now ready to be used!

C. Liveness implementation

Liveness is the idea that a system eventually makes progress, and is not completely blocked forever. Liveness properties sometimes require assumptions on the scheduler. Here, our liveness property is 'any hungry philosopher eventually eats.' and it requires to assume a fair scheduler.

1) Model: In this model, rather than using global variables to keep track of the current ticket number, now serving counter, and processes, we bundle everything into a state object. The global "Ticket System State" (Fig. 7) helps us to track everything. While the first model could only be used to easily prove the safety of the system, the second models both safety and liveness.

Functions are used to transition from one state to the next. This model also makes use of traces, which map times to states. Each trace must begin in a state satisfying the

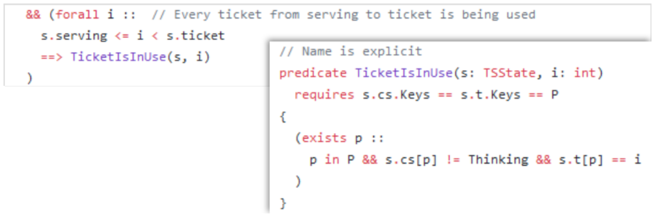


```
datatype TSState = TSState(ticket: int, serving: int, cs: map<Process, CState>, t: map<Process, int>)
```

Figure 7. Definition of our `TSState`.

conditions in the predicate Init() and each pair of consecutive states must be the result of the execution of a single atomic event by the process scheduled at the time.

2) `Valid` predicate: The predicate `Valid()`, used to show the safety of the model, is more or less the same as in the previous implementation except that it takes in a state as a parameter. The predicate does not depend on the fields of the class anymore. Rather, it depends on the fields of the TSState. We also have an additional condition that pertains to later proving liveness, namely that every ticket from serving to the most recently issued ticket is in use (Fig. 8).



```
&& (forall i ::  // Every ticket from serving to ticket is being used
  s.serving <= i < s.ticket
  ==> TicketIsInUse(s, i)
)
```

```
// Name is explicit
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
  )
}
```

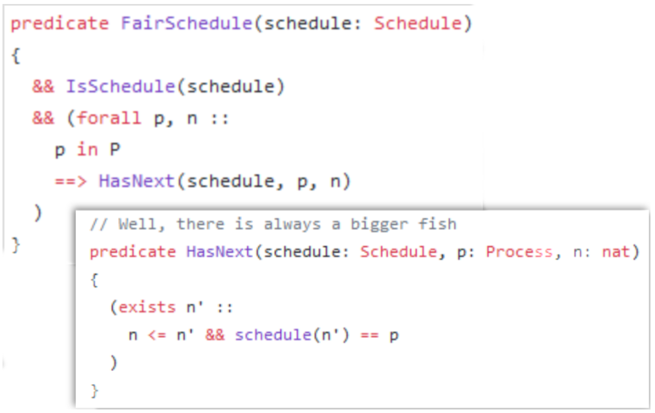
Figure 8. Every ticket from the one currently allowing to enter the kitchen to the most recently issued ticket is in use.

3) State initialisation and transition: To check the states, we will verify that the initial state is correct and that the following states are both correct and reachable from the initial state. Previous methods such as `Request(...)` are now predicates, and there are three additional predicates that are crucial to this model:

1) `Init(s)`: True if the state `s` can be a valid starting state.
2) `Next(s, s')`: True if `s'` can be reached from `s` by executing a single (valid) atomic action.
3) `NextP(s, s', p)`: True if `s'` can be reached from `s` by executing a single (valid) atomic action on process `p`.

4) Schedules and traces: But what is a trace anyway? A Trace is a function from a time to a TSState. We use traces to be able to prove properties on the evolution of the state. A Schedule is a function from a time to a Process. A fair schedule is one in which, for any process, there is some future time when the process is scheduled again.

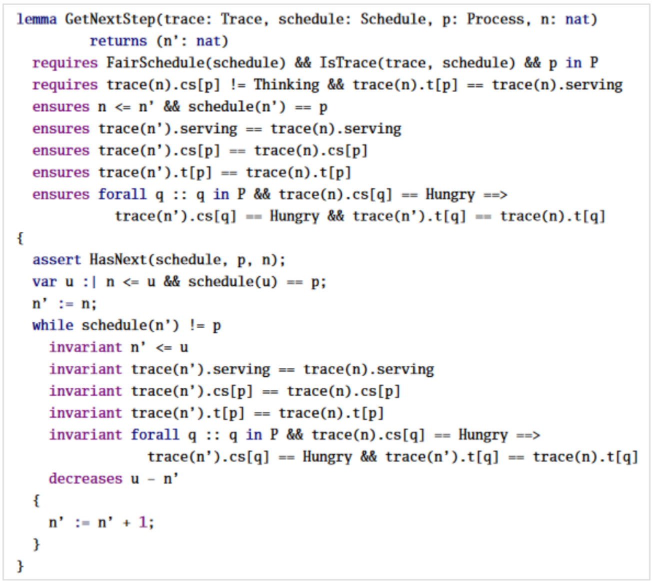
5) It's alive, it's moving, it's alive!: In Dafny, lemmas are simply methods that are not compiled into code, so they may return values and contain loops. From our definition of fairness, we know that there exists some future time in a schedule where a process `p` is scheduled. The lemma `GetNextStep()` (Fig. 10) finds and returns a future time, `n'`, where a process `p` is scheduled.



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

```
// Well, there is always a bigger fish
predicate HasNext(schedule: Schedule, p: Process, n: nat)
{
  (exists n' ::
    n <= n' && schedule(n') == p
  )
}
```

Figure 9. Fair schedule.



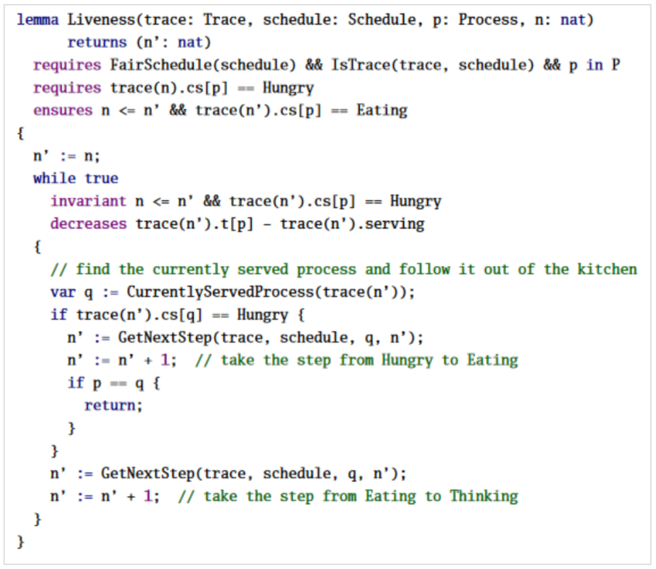
```
lemma GetNextStep(trace: Trace, schedule: Schedule, p: Process, n: nat)
      returns (n': nat)
  requires FairSchedule(schedule) && IsTrace(trace, schedule) && p in P
  requires trace(n).cs[p] != Thinking && trace(n).t[p] == trace(n).serving
  ensures n <= n' && schedule(n') == p
  ensures trace(n').serving == 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;
  }
}
```

Figure 10. The `GetNextStep` lemma.

These pieces can be used to construct the liveness lemma, which states that a hungry process eventually eats (Fig. 10). Like the previous lemma, instead of proving the existence of a future time when the process eats, the lemma simply finds and returns that time.

IV. Transactional memory in Dafny

The first part of our project introduced the concept of modelling concurrency for the purpose of program verification. The second part of the project involves applying that to model a transactional memory system. Let's start with a couple reminders. A transaction is mostly a group of read and write operations to a shared memory, and a transactional memory (TM) is a system to allow concurrent accesses to shared memory, while conserving linearizability (sometimes called atomicity) of transactions. A transaction can be either committed, which means that the transaction



```
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
  }
}
```

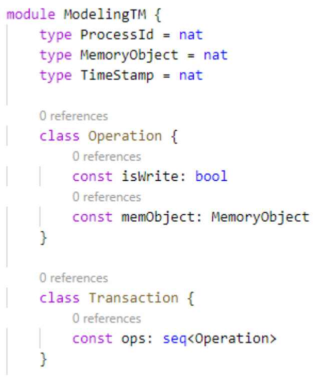
Figure 11. The final `Liveness` lemma.

succeeded while respecting linearizability, or be aborted either explicitly if the program that uses the TM system decides to cancel the transaction, or implicitly by the TM system if the system wasn't able to ensure linearizability.

A. *Simplifications*

In a real TM system, the program that uses the underlying TM system can decide the next operation of a transaction during this transaction and also decide to abort a transaction. Here, we simplify the system by defining transactions as a predefined list of read-write operations and do not consider explicit aborts. Also, when a transaction fails, a real program either decides to retry the transaction or not, but here we assume that transactions are retried until they fail.

B. *Base types*



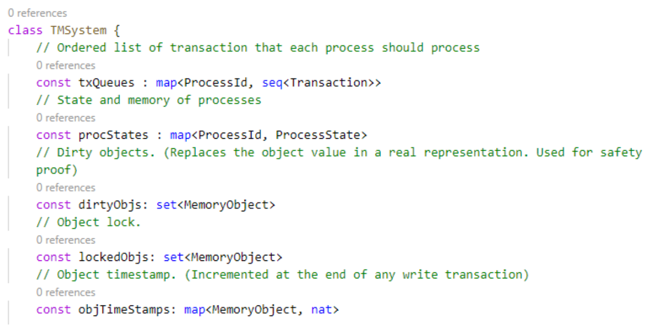
```
module ModelingTM {
    type ProcessId = nat
    type MemoryObject = nat
    type TimeStamp = nat

    0 references
    class Operation {
        0 references
        const isWrite: bool
        0 references
        const memObject: MemoryObject
    }

    0 references
    class Transaction {
        0 references
        const ops: seq<Operation>
    }
```

Figure 12. Basic types

We will first have a look at the basic types.

1) `ProcessId`, `MemoryObject`, and `TimeStamp`, are three natural numbers. The meaning of `ProcessId` and `TimeStamp` can be pretty trivially understood from their name, and `MemoryObject` represents the identifier of a region of memory (that we will call object) that has its own lock and timestamp. The actual data contained in the region is not important for the proofs as we will see later, so we do not represent it.
2) `Operation` represents one read or write operation of a transaction. Here again the value written for write is ignored, so a simple boolean to differentiate read from write operation and the identifier of the region being read/written to is enough.
3) `Transaction` represents a transaction, but due to our choices of simplifications, it is simply represented as a list of read/write operations, and the commit operation at the end is implicit.

We now have the system state.



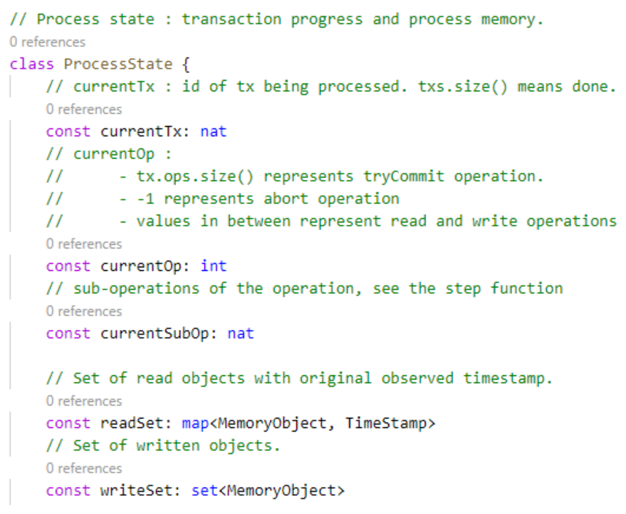
```
0 references
class TMSystem {
    // Ordered list of transaction that each process should process
    0 references
    const txQueues : map<ProcessId, seq<Transaction>>
    // State and memory of processes
    0 references
    const procStates : map<ProcessId, ProcessState>
    // Dirty objects. (Replaces the object value in a real representation. Used for safety
    proof)
    0 references
    const dirtyObjs: set<MemoryObject>
    // Object lock.
    0 references
    const lockedObjs: set<MemoryObject>
    // Object timestamp. (Incremented at the end of any write transaction)
    0 references
    const objTimeStamps: map<MemoryObject, nat>
```

Figure 13. System state object

1) `txQueues` is a mapping from each process to the list of transactions that it has to process. It is defined once at the initialisation of the system and never changes.
2) `procStates` is a mapping from each process to a `ProcessState` object, which represents the process current state.
3) `dirtyObjs` is the set of objects that were written by a yet uncommitted transaction. It is actually not part of the TM algorithm, but can more easily be used for proofs than a representations of the values contained by the memory objects.
4) `lockedObjs` is the set of all objects that are locked.
5) `objTimeStamps` represents the timestamp of each object. The timestamp is modified (incremented by 1) by a transaction that wrote something on the object, before unlocking it, in order to notify any other transaction that the object has changed in case this other transaction did not observe the locking of the object.

The object is provided with various constructors, the first one is to create an initial system from a provided `txQueues`. The other constructors represent simple "mutations” operations, creating a new `TMSystem` object with the corresponding fields modified.

Finaly we have the process state :



```
// Process state : transaction progress and process memory.
0 references
class ProcessState {
    // currentTx : id of tx being processed. txs.size() means done.
    0 references
    const currentTx: nat
    // currentOp :
    //      - tx.ops.size() represents tryCommit operation.
    //      - -1 represents abort operation
    //      - values in between represent read and write operations
    0 references
    const currentOp: int
    // sub-operations of the operation, see the step function
    0 references
    const currentSubOp: nat

    // Set of read objects with original observed timestamp.
    0 references
    const readSet: map<MemoryObject, TimeStamp>
    // Set of written objects.
    0 references
    const writeSet: set<MemoryObject>
```

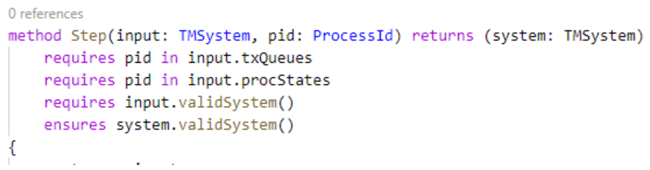
Figure 14. Process state object

1) `currentTx` is the index of the transaction being processed by the process in its queue of transactions.
2) `currentOp` is the index of the current read/write operation being run by the process in the list of operations of the transaction. The special value -1 is attributed to the abort operation and the tryCommit operation is implicitly placed at the end of the list of operations.
3) `currentSubOp` is the index of the sub-operation of the operation that we will execute next. Each sub-operation of an operation corresponds to what we consider to be an atomic operation.
4) `readSet` is the set of object that were previously read by the current transaction, associated with the corresponding originally read timestamp, and is used in the TM algorithm.
5) `writeSet` is the set of object that were previously written by the current transaction algorithm, and is also used in the TM algorithm.

As for the case of the system state, the `ProcessState` has various constructors, the first one taking no arguments and creating an initial process state with numeric value at 0 and empty sets. The other constructors are again used for “mutation” and create new instances of the object with the corresponding fields modified.

C. TM algorithm

The whole TM algorithm is defined in a function called `Step`. The `Step` function takes two arguments as input, the current system state and the id of the process to run, and returns the system state after running a single atomic step (sub-operation) of the process. The `Step` requires the input system state to be valid according to the `validSystem` function and ensures that the output system is also valid.



```
0 references
method Step(input: TMSystem, pid: ProcessId) returns (system: TMSystem)
    requires pid in input.txQueues
    requires pid in input.procStates
    requires input.validSystem()
    ensures system.validSystem()
{
```

Figure 15. The signature of the `Step` function

The algorithm we used is similar to the “algorithm 2” example provided in [3] but is not the same. It is defined as follows, with each operation being split into multiple atomic sub-operations :

Sub-ops of the write operation:

1) The first sub-operation is a try-lock : If the lock of the object is not yet acquired (the object is not in the `writeSet`), we try acquiring the lock and add the object to the `writeSet` if we succeed. If we couldn’t acquire the lock, we immediately set the next operation to be the abort operation.
2) The second sub-operation is the actual write. Since we don’t represent the values we write, we instead simply mark the object as dirty.

Sub-ops of the read operation:

1) The first sub-operation is to read the timestamp of the object and add it to the `readSet` (unless the object was already in the `readSet` or in the `writeSet`)
2) The second sub-operation is to check that the lock is not acquired by any process.

Sub-ops of the tryCommit operation:

1) In the first sub-operation, we check that the lock of every object in the `readSet` is free or owned by this process (to ensure that no other process is currently writing on those objects). If a lock happens to be locked by another process, we immediately set the next operation to be the abort operation.
2) We then validate all the timestamps of the objects in the `readSet` by verifying that they did not change. If any write operation was done by another process since we started reading this object, we would either have seen the lock locked at the previous operation or see the timestamp being changed. If the timestamps are not valid, we set the next op to be the abort operation. If all timestamps are valid, we passed all the tests and can finally be sure that the transaction we commit, so we can clear the “dirty” mark of the object in the `writeSet`.
3) We then update the timestamps of every objects in the `writeSet`.
4) And finaly, we unlock all the locks of objects in the `writeSet`. The `readSet` and `writeSet` are also

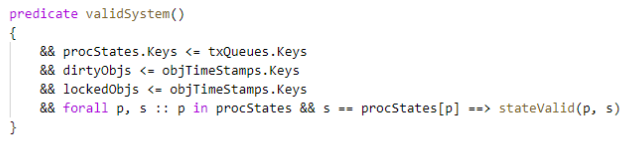
emptied as we set the next operation to be the start of the next transaction.

Since there can be many different locks and timestamps to check, increment and update, each sub-operation of the tryCommit operation would in reality represent many small atomic operation but we decided to do then all in one here for simplification, and we do the same in the abort sub-operations.

Sub-ops of the abort operation:

1) We first clear the “dirty” mark of the objects in the `writeSet`, which would correspond to the restoration of the original values stored in the memory objects if we were in a real TM system.
2) We then update the timestamp of the objects in the `writeSet`, even though we restored the values, to notify the other processes that they might have read an incorrect uncommitted value.
3) And we finally unlock the locks of objects in the `writeSet` before emptying the `readSet` and `writeSet` and setting the next operation to be the restart of this transaction.

D. Validation of the state



```
predicate validSystem()
{
    && procStates.Keys <= txQueues.Keys
    && dirtyObjs <= objTimeStamps.Keys
    && lockedObjs <= objTimeStamps.Keys
    && forall p, s :: p in procStates && s == procStates[p] ==> stateValid(p, s)
}
```

Figure 16. The `validSystem` predicate

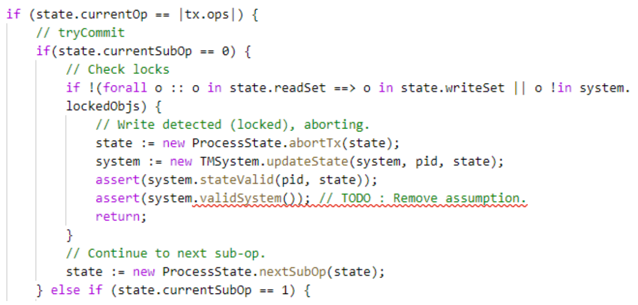
As we have seen before, we check that the system state is valid before and after each call to `Step` by using the `validSystem` predicate. `validSystem` is a method of the `TMSystem` object, and takes no argument as it can implicitly access all the fields of the system state.

The most important part of this predicate is the forall that ensures that every process of the system is in a valid state using a second predicate `stateValid` (also a method of the `TMSystem` object) which takes the process and it's state as arguments.

The `stateValid` predicate ensures various basic properties of the process state and can be extended at will. As of now, the following properties are checked :

1) the value of `currentTx` should be in a valid range for that process.
2) if the process has finished processing all of it's transactions, it should no longer execute operations.
3) if the queue is not done :
    a) the value of `currentOp` should be in the valid range for that transaction.
    b) the value of `currentSubOp` should be among the possible valid values for the current operation.
    c) `readSet.Keys` is a subset of `objTimeStamps.Keys` and `writeSet` is a subset of `lockedObjs` (which is also a subset of `objTimeStamps.Keys`).

E. Dafny limitations



```
if (state.currentOp == |tx.ops|) {
    // tryCommit
    if(state.currentSubOp == 0) {
        // Check locks
        if !(forall o :: o in state.readSet ==> o in state.writeSet || o !in system.
        lockedObjs) {
            // Write detected (locked), aborting.
            state := new ProcessState.abortTx(state);
            system := new TMSystem.updateState(system, pid, state);
            assert(system.stateValid(pid, state));
            assert(system.validSystem()); // TODO : Remove assumption.
            return;
        }
        // Continue to next sub-op.
        state := new ProcessState.nextSubOp(state);
    } else if (state.currentSubOp == 1) {
```

Figure 17. Dafny displaying assertion violation

Initially, we had planned to continue working on the transactional memory system in Dafny and prove various properties of the system, but even proving that `Step` ensures `validSystem` on the output already proved to be much more difficult than we initially suspected. While Dafny worked well in the case of the ticket system, it became hard to debug in the case of a more complex program. Dafny provides very few details about what part of the proof is failing and why. Additionally, no distinction is made between a timeout of the solver and a violation of the proofs. As an added hurdle, there is little Dafny documentation available online. For that reason, we decided to try rewriting the project in Stainless to see if the proofs are easier to do.

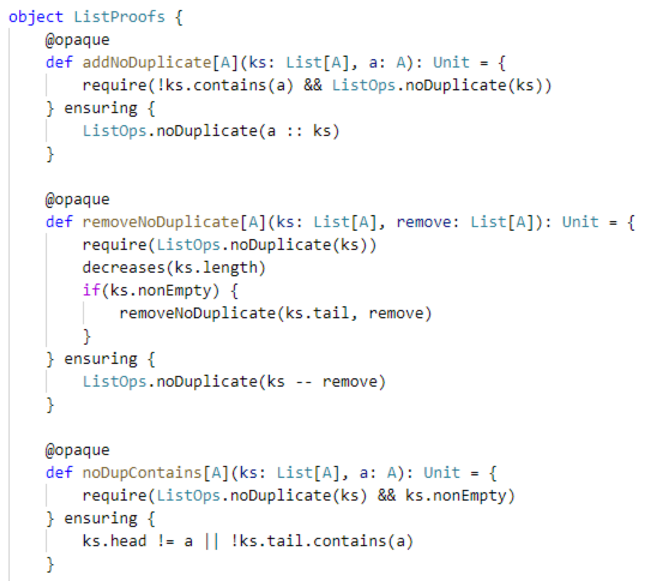
V. Transactional memory in Stainless

Stainless and Dafny are similar in that they are both based on languages that support object-oriented programming (Scala and C#, respectively) and are based on systems of preconditions and post-conditions to prove different properties of the programs. These similarities allowed for a quite intuitive rewrite of some aspects of the Dafny transactional memory, but some differences made this rewrite harder than we expected.

A. Stainless collections

While a `seq` in Dafny can easily be translated into a Stainless `List`, we had to make a choice in how we choose to translate Dafny's set and map.

Stainless has a `Set` type defined but it relies on Scala's `Set` and provides only a small set of predefined lemmas (and it seems that some of those predefined lemmas are not even true as we reported : urlhttps://github.com/epfl-lara/stainless/issues/881). Because of that, we chose to translate a `set` into a Stainless `List` that verifies the



```scala
object ListProofs {
    @opaque
    def addNoDuplicate[A](ks: List[A], a: A): Unit = {
        require(!ks.contains(a) && ListOps.noDuplicate(ks))
    } ensuring {
        ListOps.noDuplicate(a :: ks)
    }

    @opaque
    def removeNoDuplicate[A](ks: List[A], remove: List[A]): Unit = {
        require(ListOps.noDuplicate(ks))
        decreases(ks.length)
        if(ks.nonEmpty) {
            removeNoDuplicate(ks.tail, remove)
        }
    } ensuring {
        ListOps.noDuplicate(ks -- remove)
    }

    @opaque
    def noDupContains[A](ks: List[A], a: A): Unit = {
        require(ListOps.noDuplicate(ks) && ks.nonEmpty)
    } ensuring {
        ks.head != a || !ks.tail.contains(a)
    }
```

Figure 18. Some lemmas on List with no duplicates

`ListOps.noDuplicate` predicate, and build our own set of lemmas.

Then, for Dafny's map, Stainless has many options :

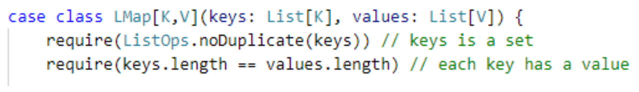
- A `Map` type which like `Set`, relies on Scala's `Map` and provides only a small set of predefined lemmas.
- A `CMap` type which represents a complete map based functions but with no defined set of keys.
- A `ListMap` type, which represents a map as a list of tuples using Stainless' `List` implementation.

This last option seemed the most relevant of the three has we needed a defined set of keys and `ListMap` provided a larger amount of predefined lemmas but instead we chose to write `LMap`, an implementation similar to `ListMap` but using two separate lists of the same size instead of a single list of tuples. The advantage of using two lists is that it makes several lemmas easier to prove, like how the set of keys (the first list) do not change when we update an existing value. Also, having our own implementation makes it simpler to define new methods on it, like the `mapValues` function we defined.

Now the thing is we needed to create many lemmas on these collections for the later proofs and it took a lot of time, as it is not always trivial (and stainless sometimes seems to encounter some bugs in the proving as we reported here : urlhttps://github.com/epfl-lara/stainless/issues/883) and we probably should have decided to define assumed lemmas on the properties of collections to focus on the actual proving of the TM algorithm.

B. validating the state in Stainless

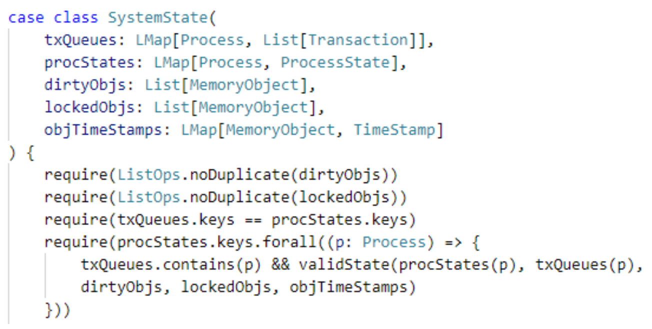
Once we rewrote the state objects in Stainless, we could start writing the `Step` function and the predicates to validate the state.



```scala
case class LMap[K,V](keys: List[K], values: List[V]) {
    require(ListOps.noDuplicate(keys)) // keys is a set
    require(keys.length == values.length) // each key has a value
```

Figure 19. LMap signature

An advantage of stainless over Dafny is that we can write require statement on the objects we define (but be careful when using those as it seems that the solver doesn't know when to timeout on their proof when we start creating new instances of those objects), which allows to avoid the repetition of require statements on every function that uses that object and of ensure statements on every function that creates instance of that object. Since Stainless offers that feature, we started implementing the validation of the state before even writing the `Step` and as we tried to prove that the state remains valid after a call to any of the "mutation" functions, but we started running out of time as every proof was requiring more and more underlying lemmas on the collections and the proofs started to be less evident.



```scala
case class SystemState(
    txQueues: LMap[Process, List[Transaction]],
    procStates: LMap[Process, ProcessState],
    dirtyObjs: List[MemoryObject],
    lockedObjs: List[MemoryObject],
    objTimeStamps: LMap[MemoryObject, TimeStamp]
) {
    require(ListOps.noDuplicate(dirtyObjs))
    require(ListOps.noDuplicate(lockedObjs))
    require(txQueues.keys == procStates.keys)
    require(procStates.keys.forall((p: Process) => {
        txQueues.contains(p) && validState(procStates(p), txQueues(p),
        dirtyObjs, lockedObjs, objTimeStamps)
    }))
```

Figure 20. require statements on the SystemState case class

Unfinished Work: Some parts of the initial Dafny program were somewhat more difficult to port to Stainless so, due to time constraints, were left unfinished. These parts appear in the file titled `simple_tm_continued_work.scala`. In particular, the file features implementations of `stateValid` (which is completed but not used anywhere in the code) and `validSystem`. The latter, as noted in the code, is incomplete due to the combined difficulties of using Stainless lists and BigInts, making iterating through a Stainless list difficult either due to `foreach` not existing for Stainless lists or Stainless not being able to detect a decreasing measure if a `while` loop is used. Both of these functions would be used in the `Step` method,

VI. Future Work

Due to the difficulties we encountered using Dafny, we were limited in the amount of work we could do with our Stainless re-write. As a result, continuing the project could entail implementing a Stainless version of the `Step` method, proving more properties about the simple transactional memory algorithm in Stainless, and implementing other, more complex transational memory algorithms and proving properties about those.

VII. Conclusion

Our initial interest in completing this project was both to learn a new program verification tool to expand our skills and knowledge relating formal verification and also to try to use those verification tools to prove properties of a more complex and meaning-full system that has a concrete real word use. For the first part of this goal, the project was very successful. We learned how to use Dafny and recreated the ticket system example that was in the Leino paper. The project was less successful in the implementation and proving of a complex system like a TM but it helped us learn limitation of the current existing tools and how complex formal verification can be when you work on complex systems and do not rely on any assertion over the underlying objects you use (like collections).

Acknowledgements

We would like to thank Viktor Kuncak, as well as the whole pedagogical team, Jad Hamza, Romain Edelmann, David Resin, for teaching and guiding us in the first steps of this project.

We would also like to thank our friends and colleagues for their help and proofreading of the various presentations and reports.

References

[1] K. R. Leino, "Dafny Tutorial." [Online]. Available: https://rise4fun.com/Dafny/tutorial

[2] ——, "Modeling Concurrency in Dafny," in SETSS, 2017.

[3] R. Guerraoui and M. Kapalka, "The semantics of progress in lock-based transactional memory," in POPL '09, 2009.

[4] M. Fu, Y. Zhang, and Y. Li, "Formal verification of concurrent programs with read-write locks," Frontiers of Computer Science in China, vol. 4, pp. 65–77, 03 2010.

[5] E. Guerrant, C. Burgelin, and H. Hueber, "Concurrency and Transactional Memory in Dafny," 2021. [Online]. Available: https://github.com/Zyfarok/fv2020-concurrency-and-tms

[6] Wikipedia, "Short-circuit evaluation," http://en.wikipedia.org/w/index.php?title=Short-circuit%20evaluation&oldid=995172145, 2020.