SAND2016-4937C

Verification by way of refinement: a case study in the use of Coq and TLA in the design of a safety critical system

Philip Johnson-Freyd[1], Geoffrey C. Hulette[2], and Zena M. Ariola[1]

[1] University of Oregon, Eugene OR
{philipjf,ariola}@cs.uoregon.edu
[2] Sandia National Laboratories, Livermore CA
ghulett@sandia.gov

Abstract. Sandia engineers use the Temporal Logic of Actions (TLA) early in the design process for digital systems where safety considerations are critical. TLA allows us to easily build models of interactive systems and prove (in the mathematical sense) that those models can never violate safety requirements, all in a single formal language. TLA models can also be *refined*, that is, extended by adding details in a carefully prescribed way, such that the additional details do not break the original model. Our experience suggests that engineers using refinement can build, maintain, and prove safety for designs that are significantly more complex than they otherwise could. We illustrate the way in which we have used TLA, including refinement, with a case study drawn from a real safety-critical system. This case exposes a need for refinement by composition, which is not currently provided by TLA. We have extended TLA to support this kind of refinement by building a specialized version of it in the Coq theorem prover. Taking advantage of Coq's features, our version of TLA exhibits other benefits over stock TLA: we can prove certain difficult kinds of safety properties using mathematical induction, and we can certify the correctness of our proofs.

1 Introduction

Sandia Laboratories builds extremely high consequence systems. Logical errors in these systems could incur enormous costs both financially and in loss of life. It is of course natural then to want to apply formal methods to verify such systems. However, historically, the use of formal methods in digital system design at Sandia has been limited. To rectify this, Sandia engineers are implementing a new methodology to integrate formal methods into the specification and design phase of high consequence systems. However, we need specification languages which comport with the way designers think about the systems they are designing and which are easy enough to use for engineers who are not formal methods experts. At the same time, we need tools which make verification tractable at scale.

Sandia engineers have begun to make use of the Temporal Logic of Actions (TLA) [9] for formalizing specifications. TLA has proven itself to be an effective formalism for describing the evolution of digital systems over time. However, TLA$^+$ and its associated tooling still faces limitations. TLA$^+$ comes with a model checker [17], but this model checker, while fully automated, is insufficient for proving many properties. Often times engineers need to work over infinite state spaces and want to verify properties beyond the capability of any model checker. Such properties require instead interactive theorem proving with human supplied lemmas and inductive hypotheses. What is more, model checkers are complex pieces of software and could contain bugs. We want to be able to independently verify claims made by verification tools. In short, we want proofs and we want those proofs to be easily, and independently, machine checkable.

In order to overcome some of these limitations we have embedded a version of TLA inside the proof assistant Coq [7]. Coq is a highly advanced tool, however, its logic is not specialized to our application domain. Further, Coq's flexibility comes at the expense of usability as it is a tool which is geared primarily to formal method experts. We needed a way to leverage Coq's power while retaining TLA's ease of use and this has driven the design of our embedding, TLA^{Coq}.

The initial problem which motivated the design of TLA^{Coq} is the specification of a component of a high consequence digital system produced by Sandia. We initially developed a formal model of this component, called the Arbitrary Waveform Generator (AWG), in TLA^{+}. After running into limitations with TLA^{+} we developed TLA^{Coq} and transitioned the AWG model to our embedding. Doing so allowed us to prove properties which we could otherwise only partially verify by way of bounded model checking. It also enabled us to adopt a development approach where we composed orthogonal refinements to construct a complete model. The compositional approach is crucial: different refinements reveal different aspects of the system, we need to be able to work with refinements individually or in combination while managing complexity. We believe even larger gains will appear as we apply these techniques to bigger systems.

In Section 2 we describe the high level specification of the Arbitrary Waveform Generator. In Section 3 we discuss our initial attempts to formalize this specification in TLA^{+} and the challenges we encountered. In Section 4 we describe how we use our embedding TLA^{Coq} and how we structured the AWG development in it. In Section 5 we cover the design decisions and technical details of our embedding. We conclude with lessons learned.

2 Application: Arbitrary Waveform Generator (AWG)

The Arbitrary Waveform Generator (AWG) is a component of a high consequence digital system being developed at Sandia. The AWG is used for storing "patterns" in memory which are later played out as timed waveforms. While relatively simple, the AWG component is a real circuit that is being incorporated into silicon in production. Its specification presented in this paper was developed as a collaboration between formal methods experts and domain engineers with an eye towards more broadly introducing certain formal methods techniques at Sandia. This process of collaboration proved helpful early on in clarifying details of the AWG's original requirements document that otherwise might have been missed. And, the act of formalizing the resulting specification revealed weak spots and avenues to improve our formal method techniques.

Our ultimate application domain demands a very high level of assurance in order to avoid loss of life. However, there is nothing intrinsic about the AWG that is unique to our application domain. Indeed, similar systems are likely to be used in a host of

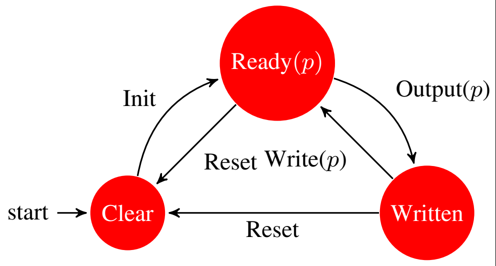

Fig. 1. Simplified AWG state machine

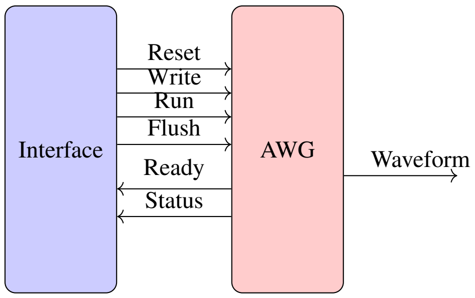

Fig. 2. AWG logical interface

applications. The timed input/output behavior of the AWG will appear in any system where precise playback of programmed in patterns is required and so could be used for anything from a musical alarm clock to an autonomous vehicle. Given both the generality of the AWG and its importance in a concrete high consequence system we believe it serves as a suseful demonstration for digital system formalization.

The AWG needs to support two main operations. The first is to read in a pattern from its input and store that pattern in its memory. The second is to, upon receiving a special signal, begin playing out the value in its memory. At any time the AWG can also be “reset" by passing in a certain signal, clearing its memory. Thus, the AWG can be conceptually thought of as a state machine (see Figure 1) that can be in one of three modes: initially, or upon being reset, the AWG will not have a meaningful value in its memory and thus not be able to be played. From the cleared state, the AWG can receive an input pattern which it will encode into memory leaving it ready to play out the pattern. Finally, the AWG can play out the pattern stored in its memory. This process of playing out the pattern will happen in a timed manner, removing bits from its pattern buffer as they are output, and so eventually leaving the AWG with its buffer erased. From either the “Ready" or “Written" states the AWG can be “reset" returning it to a “Clear" state.

This description of the AWG is, of course, much more abstract than what would suffice to describe an implementation. For example, it does not include the intermediate states that will be encountered as the pattern is played out in a timed manner or while initiating the machine with a new pattern. We will therefore need to *refine* this specification. But, it is worth emphasizing that a formalization of the specification at this level of abstraction also is insufficiently detailed for those building components which interact with the AWG. The protocols a component uses to communicate with other components are very relevant to the design of those other components. Nonetheless, it is still important to have conceptual models, and we believe formal models as well, which operate at these higher levels of abstraction. This leads to an important observation: refinement of component specifications matters not just in the design of individual components but in systems of components as well. We must refine protocols in addition to state machines. And, this is necessary to build tractable high level specifications.

Thus we will refine our model to clarify that the pattern is played out not all at once but as a sequence of states, requiring an internal memory. Additionally we need to refine our model to describe the channels on which it communicates (See Figure 2). One of these channels will carry signals we will call commands and will take one of three forms: a “Run" command instructing the AWG to play out its memory, a “Write" command containing a pattern, or a “Flush" command asking the AWG to be flushed. The other input channel is a simple wire used for sending “Reset" signals. There are three output channels: the waveform lines on which the pattern will be played, the “Ready" line, and a “Status" line used for such things as error codes. The AWG is only expected to handle commands while it reads as having a “Valid" status and a true “Ready" line. And, we would expect it to take multiple clock cycles for initiation after receiving a command or resetting before it was ready to accept another command. However, it should be able to “Reset" at any time. This then illustrates the difference between the “Flush" command and the “Reset" signal–while ultimately serving a similar purpose, the “Flush" command will only come when our system is ready to receive commands and so can take advantage of this stronger precondition in its implementation while a “Reset" must be possible in any state of the system.

The addition of memory and timed output is mostly independent from the issue of the communication protocol. When we think about the details of one refinement of the initial basic model we need not think about the details of the other.

3 Expressing the AWG in TLA$^+$

In order to formalize the AWG system we turned to the Temporal Logic of Actions (TLA). TLA has desirable properties for formulating a system like the AWG. Foremost, TLA specifications are not sensitive to the rate of the passage of time, i.e. it is stuttering invariant [3]. This enables the development of formal TLA specification using a process of refining more abstract specifications into more concrete ones in much the same way we approach designs like the AWG informally [6]. TLA takes a logic centric view: specifications and theorems about specifications are both just logical formulae and the statement that one specification refines another is interpreted simply as that the more refined specification implies the more abstract one, perhaps along some “refinement mapping” of their underlying state spaces [1]. Stuttering invariance implies that refinements can in many instances *slow down time* through the addition of intermediate states in the refined specification which are not observable in the more abstract one.

3.1 Models and Refinements

We took a refinement based approach to formalizing the AWG in TLA$^+$. Our initial specification, therefore, only modeled some very basic properties of the AWG system. The model (Figure 3) is parameterized by a set called `Pattern` which serves to abstract away the details of the patterns stored in memory and played out by the AWG. The constant `NilPattern` represents the “empty” pattern. The memory status following self-check may be `Valid` or `Invalid`. The specification variable `memory` is a `Pattern` representing the state of memory. The variables `ready`, `status`, and `output` represent the output lines.

```
Next ≜ Initialize
       ∨ Reset
       ∨ Flush
       ∨ ∃p ∈ Pattern.Write(p)
       ∨ Run

Write(p) ≜ ready = TRUE
         ∧ status = Valid
         ∧ IF p ≠ NilPattern
             THEN memory′ = p
             ELSE UNCHANGED memory
         ∧ UNCHANGED << ready,
             status, output >>

Run ≜ ready = TRUE
    ∧ status = Valid
    ∧ IF memory ≠ NilPattern
        THEN
          output′ = memory
          memory′ = NilPattern
        ELSE UNCHANGED << output,
          memory >>
    ∧ UNCHANGED << ready, status >>

Reset ≜ ready′ = FALSE
      ∧ status′ ∈ MemStatus
      ∧ memory′ ∈ Pattern
      ∧ output′ = NilPattern

          vars ≜<< ready, status, memory, output >>
Fairness  ≜  WF_vars(Initialize)
    Spec  ≜  PowerOn ∧ □[Next]_vars ∧ Fairness
```

Fig. 3. Excerpt of Basic Model in TLA$^+$

The `Next` relation is then defined as the conjunction of five actions. These actions specify relations between a current and a next state. In TLA the apostrophe at an end of a variable means it refers to the next value of that variable and can only occur in an action. $\mathtt{WF}_{\mathtt{vars}}(A)$ is the TLA formula which asserts weak fairness for the action A and means that A either keeps happening or is eventually impossible. $\Box[A]_{\mathtt{vars}}$ means that at every step A happens or the value of `vars` is unchanged.

Some desired properties of this specification are shown in Figure 4. The notation $\leadsto$ is pronounced "leads to" [11] and is an example of a liveness property. Namely, that `ready` keeps becoming `TRUE`. Thus, the AWG will keep accepting and playing out new patterns and so we can rule out whole classes of anomalous and potentially dangerous behavior such as playing out one waveform forever.

One of the areas of missing details in the Basic model is in the handling of commands. In the basic model, `Run`, `Flush`, and `Write`(p) are treated simply as events. However, we know that they are actually commands which come as input to the system and might take multiple steps to handle. Therefore, our first refinement is to incorporate the more detailed notion of commands.

In the new model our state consists of the four variables from the Basic model plus variables `cmd` and `ctrl`. The idea is that `cmd` will store the command the machine is currently processing while `ctrl` will store its status in that process.

$$\texttt{THEOREM Spec} \Rightarrow \Box\texttt{TypeInvariant}$$

$$\texttt{CommandsEnabled} \triangleq \texttt{ENABLED Run} \land \texttt{ENABLED Flush}$$
$$\land \forall p \in \texttt{Pattern}.\texttt{ENABLED Write}(p)$$

$$\texttt{THEOREM Spec} \Rightarrow \Box\texttt{CommandsEnabled}$$

$$\texttt{THEOREM Spec} \Rightarrow \Box(\texttt{ENABLED Reset})$$

$$\texttt{THEOREM Spec} \Rightarrow (\texttt{ready} = \texttt{FALSE} \leadsto \texttt{ready} = \texttt{TRUE})$$

Fig. 4. Properties of Basic Model in TLA$^+$

$$\begin{aligned}\texttt{DoFlush} &\triangleq \texttt{cmd.op} = \texttt{Flush}\\ &\quad \land \texttt{memory}' = \texttt{NilPattern}\\ &\quad \land \texttt{output}' = \texttt{NilPattern}\\ \texttt{Next} &\triangleq \texttt{Initialize} \lor \texttt{Reset}\\ &\quad \lor \texttt{Do} \lor \texttt{Resp}\\ &\quad \lor \exists c \in \texttt{Command}.\texttt{Req}(c)\\ \texttt{Spec} &\triangleq \texttt{PowerOn} \land \Box[\texttt{Next}]_{vars} \land \texttt{Fairness}\end{aligned}$$

$$\begin{aligned}\texttt{Do} &\triangleq \texttt{ready} = \texttt{TRUE}\\ &\land \texttt{status} = \texttt{Valid}\\ &\land \texttt{ctrl} = \texttt{Busy}\\ &\land \texttt{DoFlush} \lor \texttt{DoWrite} \lor \texttt{DoRun}\\ &\land \texttt{ctrl}' = \texttt{Done}\\ &\land \texttt{UNCHANGED}\\ &\quad << \texttt{ready}, \texttt{status}, \texttt{cmd} >>\end{aligned}$$

Fig. 5. Excerpt from Command Module

In place of the actions Flush, Run, and Write(p) from the Basic model, we have a more complicated implementation of commands by way of the actions Do, Resp, and Req(c). A client issuing a command is represented by Req(c) actions. The AWG processes the command and updates its internal data structures in a Do action. Finally, the AWG resets its interface to accept further commands in the Resp action. These actions are defined such that the desired order of operations is strictly enforced and, moreover, that the correspondence with the Basic model is maintained.

The new model allows us to define properties related to commands. We also want to ensure that what we have described really is a refinement of the Basic model, so that all the properties we established about the Basic model hold automatically in the Command model. Formally, we must demonstrate that our new Spec formula implies the Spec from the Basic model.

In addition to the more detailed theory of commands, we must consider the playing of patterns in more detail. The AWG should not play patterns to the output all at once, but rather play them slowly. In order to handle this we can modify our original basic specification to store the memory not as a single element of the abstract set Pattern but rather as a finite sequence of elements from such an abstract set. Playing a pattern will now be a multi-step process so we also add a variable mode, the type invariant for which will be that it is a member of the set containing only the constants Record and Play. The Flush and Write actions are modified to only fire when the mode is in Record (which they also preserve).

Playback is now implemented with three events. The StartPlay event can occur when the mode is Record, the status is Valid, and the ready flag is TRUE. It requires the new state have the mode Play but is otherwise unmodified. The DoPlay event can only fire when the status is `Valid`, the `ready` flag is `TRUE`, and has the effect of popping the first element of the memory sequence into the output leaving everything else unchanged. `EndPlay` becomes possible instead of `DoPlay` when the `memory` is the empty sequence and simply switches the `mode` back to `Record`.

3.2 Limitations of the TLA$^+$ framework

At this point we have described a set of TLA$^+$ models, each describing a different aspect of the AWG. We have extended the Basic model in two different ways incorporating different aspects of the full system we care about. We have also stated, and model checked, a number of properties. We would therefore like to complete the picture by combining all the various aspects of the three models into a single complete model. We would also like to verify the various correctness theorems we have stated.

However, we encounter challenges in both these goals. Both the Command model and the Memory model were developed by extending the Basic model, but in doing so we had to restate essentially the entire model. It would be undesirable indeed to have to fully write out yet another model. Instead, we would like to simply be able to assert that our full specification is exactly the refinement of the basic model which extends the Basic model in the way the Command model does and the way Memory model does.

We are able to use the TLA$^+$ model checker to check the theorems we have stated, but only by first instantiating the abstract set pattern with a concrete, finite, set. The model checker tells us that our theorems are true, but only for this concrete set. How can we be sure they hold for *any* instantiation of `Pattern`? Moreover, the model checker will claim that the properties hold, but it does not provide any sort of witness or reason as to why this is true. Instead, we gain confidence in correctness only relative to our confidence in the correctness of the model checker. While a TLA$^+$ proof system (TLAPS) [5] which might help rectify some of these issues has been partially developed it is incomplete and does not currently support temporal reasoning.

4 Expressing the AWG in TLA$^{\text{Coq}}$

We want to be able to prove properties about our models using inductive reasoning. Moreover, we want to support parametric reasoning — it should be possible for us to consider some aspects of a model as abstract parameters and still be able to prove properties about that model which hold for any instantiation of those parameters. Further, we want proof witnesses and a small trusted base so we do not have to depend on the correctness of complex pieces of machinery like model checkers. Moreover, we would like a framework which not only provides these things but which also has good support for “programming in the large" and building abstractions.

Interactive theorem provers based on type theory excel at precisely this point. By using a small core logic they separate the problems of finding proofs and interacting with the system with the problem of checking a proof already found. Systems like Coq and Isabelle [16] come equipped with fully powerful fully automated methods (such as Coq’s Presburger solver which uses the Omega test [12]) as well as tools for developing

```
Section Model.
  Variables hr hr' : Z.

  Definition Init := 1 <= hr <= 12.
  Definition Next := hr' = hr mod 12 + 1.
End Model.

Definition Spec := `Init `/\ [][Next].
```

Fig. 6. A model of a clock in our embedding of the TLA in Coq

new domain specific automation, but these capabilities are kept separate from the logic kernel.

Coq in particular seems like a close match to our needs. Its type theory not only extends higher order logic with dependent types, it also allows for quantification over universes providing the ability to abstract over types. Moreover, Coq has a rich module system, and the Coq tool has proven to scale to very large scale projects such as a fully certified C compiler [4] and the proof of the four color theorem [8]. However, unlike what we see with TLA$^+$, Coq's logic is not geared specifically toward expressing systems which evolve over time. While TLA is well suited to describing systems like the AWG, vanilla Coq is not.

TLA$^{\text{Coq}}$ is an embedding of TLA into Coq. It allows us to take advantage of Coq's features which make scalable verification tractable while presenting an interface similar to TLA. Moreover, while Coq is a highly advanced tool requiring a great deal of time to master, one of our goals on the AWG project has been to support collaboration between subject matter experts and system designers with formal methods experts. As such, we have aimed to make the embedding relatively straightforward to use even for non experts. For example, we can use our embedding to formalize a model of a simple clock akin to that considered by Lamport [10, 11]. The Coq code in Figure 6 provides a complete specification of the clock using our library. The `Init` and `Next` definitions inside the `Model` section describe the initial configuration and evolution of the state of the clock which we encode as a single integer `hr`. `Init` requires this variable have a value between one and twelve. `Next` relates the variable `hr` to the variable `hr'` (representing the next time) whenever `hr'` is equal to one plus the value of `hr` modded out by twelve. These definitions are simply predicates in Coq and so involve no temporal operators. However, the definition of `Spec` below them occurs within the embedded TLA. `Spec` is simply the temporal logic conjunction of the requirement that `Init` holds for the initial state and that every future change in state happens according to the `Next` predicate. Note the use of '$\wedge$ instead of $\wedge$ in the definition of `Spec`. This is because we are constructing the conjunction of two TLA formulae and not the meta logical conjunction of two Coq propositions. Similarly, the quote in '`Init` lifts `Init`, a Coq predicate, into a TLA formula. While exceedingly simple, the clock already demonstrates the main features we need to specify the AWG. In particular, we can not only state properties of interest about the Coq model, but prove them as well. For example, an important property we can prove about the clock is that the value of `hr`

```
Theorem hour_inv : valid (Spec `=> [] `Init).
Proof.
  unfold Spec, Init, Next.
  apply tla_inv.

  (* Base case *)
  intuition.

  (* Inductive case *)
  intros.
  assert (0 <= x mod 12 < 12) by (apply Z.mod_pos_bound; intuition).
  intuition.
Qed.
```

Fig. 7. Proof of Type Invariant for the Clock

will always be between one and twelve. The proof in Figure 7 of this property demonstrates a standard style of proofs in our system, where TLA specific reasoning tools and theorems are used to handle the temporal backbone of formulae but standard Coq tactics are used for the non-temporal leaves. In this case we need to instruct Coq to prove our property by induction. Then, a fully automated method (`intuition`) becomes sufficient for handling the base case. The inductive step is almost fully automated, but not quite, requiring a small amount of guidance and appeal to a general mathematical theorem from Coq's standard library. The availability of theorems such as these is one of the benefits of working with Coq.

Using TLA$^{\text{Coq}}$ we formulated the specification of the AWG module. Following the original version in TLA we approached this through a series of modules corresponding to the Basic, Command, and Memory refinements. As in the original Basic specification we took the set of patterns to be abstract. We describe the parameters to the Basic mod-

```
Module Type BasicParams.
  Parameter PatternIsh : Type.
  Parameter IsPattern : PatternIsh -> Prop.
  Parameter NilPattern : PatternIsh.
  Parameter NilPatternPattern : IsPattern NilPattern.
  Parameter EqNilPatternClassical : forall p,
       IsPattern p -> p = NilPattern \/ p <> NilPattern.
End BasicParams.

Module Basic (Params : BasicParams).
  Import Params.
  Hint Resolve NilPatternPattern.
```

Fig. 8. Basic AWG Module Structure

```
Inductive MemStatus : Type := Valid | InValid.

Record St := {
    ready : bool; status : MemStatus;
    memory : PatternIsh; output : PatternIsh}.
```

Fig. 9. Basic State Type

```
Section Model.
  Variable st st' : St.

  Let ready' := (st').(ready). Let ready := st.(ready).
  Let status' := (st').(status). Let status := st.(status).
  Let memory' := (st').(memory). Let memory := st.(memory).
  Let output' := (st').(output). Let output := st.(output).

  Definition TypeInvariant :=
   IsPattern memory /\ IsPattern output.

  (* action definitions elided *)
  Definition Next :=
    Initialize \/ Reset \/ Flush \/ Run
    \/ (exists p, IsPattern p /\ Write p).
End Model.
(* definition elided *)
Definition Spec := `PowerOn `/\ [][Next] `/\ Fairness.
```

Fig. 10. Outline of Basic Model in Coq

ule (Figure 8) by way of a module type `BasicParams` which has a type representing patterns.

We defined our states as coming from the type `St` containing a memory, the output, a boolean ready flag, and the status. The status variable is defined as coming from the type containing only the constants `Valid` and `InValid`. Using this definition we can then define the various events of the model in much the same way as we encountered in pure TLA. For example, the `Next` event is defined simply as the conjunction of smaller events in Coq. To express these events in a clear way we define a `Section` and using the `Variable` keyword to lambda abstract over the the current and next states in the section. Then, we can give names to the variables as projections into the states as can be seen in Figure 10. On the other hand, the definition of the whole specification is an element of our `expr` data type and does not live in the model section. Observe that with the definition of `St` much of what we considered a “type invariant property" of the model in the pure set theoretic model will now hold automatically. For example, that `status` is in the type `MemoryStatus` is definitionally true rather than being a theorem which must be proved about the specification. However, because we encode

```
Inductive Control : Type := Ready | Busy | Done.
Inductive Command : Type :=
  | Flush : Command
  | Write : PatternIsh -> Command
  | Run : Command.
Definition IsCommand (c : Command) : Prop :=
    match c with
    | Write p => IsPattern p
    | _ => True
    end.
```

Fig. 11. Start of Command Module

membership in the pattern set as a predicate, we still have a type invariant that simply states that output and memory remain valid patterns. However, by switching to the Coq representation we can now *prove* that the type invariant always holds as an inductive property.

```
Theorem Spec_then_TypeInvariant :
     valid (Spec `=> [] `TypeInvariant).
```

And, indeed, we can state and prove the other properties from the TLA$^+$ version of the specification.

After the Basic model we construct the two refinements. The Command model works similarly to the Basic model. It is again constructed as a module functor parameterized by a module of the `BasicParams` type. Using Coq's inductive type system makes it easy to express the type of the control variable as well as the type of commands. But, we only consider a `Write` command to be valid if the stored pattern data is, in fact, a pattern. We can express this by defining a predicate on commands by pattern matching. Beyond this, the definition of the command module works as direct translation of our early TLA$^+$ specification following the same approach as we used for translating the Basic module.

The last module to translate is the memory module. Here we have as parameters not merely a base type of patterns and its associated properties, but also a maximal length of the vector of these which will be stored in memory. To represent this we use a new module type which extends the `BasicParams`.

The main idea then is to instantiate the Basic module with a representation in which patterns from the Basic module correspond to sequences of patterns in the Memory module. We do this by way of a module functor as in Figure 12. As before the Coq code is mostly a direct translation of the original TLA$^+$ specification. The refinement mapping from Memory to Basic, however, depends on instantiating Basic with a *different*, and more specific, set of parameters.

Now, we have constructed two modules in Coq which each refine our original Basic specification. We want a specification which incorporates the details from each. We would like to be able to do this in a general way: we should not have to fully describe the combined refinement at the same level of detail as the others, but rather simply declare it as the combination of the memory and command refinements.

```
Module ParamsFromMem (Params : MemParams) <: BasicParams.
  Definition PatternIsh := list Params.PatternIsh.
  Definition IsPattern := Params.IsListPattern.
  Definition NilPattern : PatternIsh := nil.
  Theorem NilPatternPattern : IsPattern NilPattern.
  Proof.
    unfold IsPattern, NilPattern. simpl. auto.
  Qed.
  Theorem EqNilPatternClassical : forall p, IsPattern p
      -> p = NilPattern \/ p <> NilPattern.
  Proof.
    intros p H.
    unfold NilPattern, IsPattern.
    destruct p. left; auto. right; intros F; inversion F.
  Qed.
End ParamsFromMem.
```

Fig. 12. Memory Module Functor

```
Record St := {
     ready : bool; status : MemStatus; mode : Mode;
     memory : list PatternIsh; output : list PatternIsh}.
```

Fig. 13. State type for Memory Refinement

One option, available to us from TLA$^+$, is to simply combine the two models as a conjunction [2]. However, a complexity arises in that the two models have different types to represent their states and we need to be explicit about what states we are using. Using the product of the two state spaces would lead to a model, but not the one we intend. Specifically, we would have, for example, two separate variables encoding the status.

Instead we construct the combined specification as the conjunction of the two refinements restricted to the case where they map to the same thing in Basic. The details are provided in an appendix.

5 Implementing TLA$^{\text{Coq}}$

There are a number of ways to use a meta logic as rich as Coq to host another logic or language. Often when considering embedded languages we contrast so-called “shallow" and “deep" embeddings [14]. In a shallow embedding each construct of the target language is directly mapped to a construct of the meta language. By contrast, in a deep embedding the target language constructs are considered as data representing syntax. Existing work includes examples of both deep and shallow embeddings of TLA into Coq. In [15] a shallow embedding of TLA in Coq was constructed to prove a meta result that all TLA specifications of a certain form satisfy a “machine closed" property

```
Module FROMMEM := ParamsFromMem (Params).
Module BASIC := Basic.Basic (FROMMEM).
Definition refinement_mapping (st : St) : BASIC.St :=
    match st with
      | {| ready := ready0; status := status0;
           memory := memory0; mode := mode0;
           output := output0 |} =>
         {| BASIC.ready := ready0;
            BASIC.status := status0;
            BASIC.memory := match mode0 with
                | Record => memory0
                | Play => ((List.rev output0) ++ memory0)
              end;
            BASIC.output := match mode0 with
                | Record => List.rev output0
                | Play => nil
              end |}
end.
```

Fig. 14. Memory Refinement of Basic

```
Theorem tla_inv_gen :
  forall (A : Type) (Init : Predicate A) (Next : Action A)
  (F P : Expr A), (valid (`Init `/\ [][Next] `/\ F `=> P)) ->
    (forall s, eval P s -> Next (s 0) (s 1) -> eval P (s @ 1))
    -> valid (`Init `/\ [][Next] `/\ F `=> [] P).
```

Fig. 15. Induction Rule

in which the liveness part of the specification does not constrain the reachable states of the system. As such, TLA formulae are interpreted simply as predicates on infinite sequences of states. A deep approach to embedding a TLA like logic in Coq was used more recently as part of the VeriDrone project [13]. VeriDrone combines discrete digital components with continuous physical ones in a hybrid cyber-physical system, and so their logic combines continuous time with the discrete transition semantics of TLA. Following Lamport, they use a two step approach where their logical syntax corresponds to a full set of "RTLA" formulae of which the stuttering invariant TLA formulae are only a subset.

We take what is mostly a deep embedding approach to the handling of TLA formulae in Coq. The set of TLA expressions is encoded as an inductive datatype parameterized by the type of states . The advantage of our design is that it yields simpler proofs for safety properties by ensuring that all formulae are stuttering invariant by construction. Safety properties are demonstrated via the induction rule in Figure 15 without having to think about stuttering steps. On top of this data type we use Coq's notations facilities to build a more convenient syntax for writing TLA formulae. We rely on Coq's extensive standard library for the mathematical language used in formulae, rather than going through the trouble of embedding set theory.

Coq has a very flexible and configurable parser, which we use to provide a convenient syntax for writing TLA formulae. This allows our Coq code to look a lot like regular TLA$^+$.

Instead of encoding TLA's proof theory directly, we provide a semantic account of the truth of TLA formulae in Coq and then we use Coq to prove the truth of a formula. Following Lamport [9], TLA formulae are interpreted as being predicates over infinite sequences of states called behaviors. Our implementation "compiles" an `Expr` value to a Coq function from behaviors to Coq's `Prop` type of propositional values. From there, proofs are done in Coq as per usual, and in particular may use Coq's standard automation facilities.

6 Conclusions

Our use of Coq to embed a TLA-like language yielded several practical benefits. Using refinement for composition, we were able to construct a model of a critical system component in stages, adding detail as we went. The use of Coq allowed us to construct proofs of correctness properties using an interactive theorem proving approach, combining automation where possible with human insight where necessary. Unlike approaches based on bespoke model checkers, Coq also lends a very high level of confidence to our proofs.

The designers and engineers involved in the specification and verification of the AWG component found the effort to be worthwhile. Using a TLA-style specification forced us to consider details about how the high-level design worked early on. Without it, we probably would not have considered those details until they were encountered by programmers or revealed in testing, at which point changing the design would have been much more difficult. We estimate that the cost of performing the formal specification and model checking was justified by the time saved later in the process, and that those costs will decrease in the future as we gain experience with the tools and techniques.

The use of interactively-developed proofs rather than automated model checking comes at a cost in terms of development time and required expertise. Improved automation within Coq would be helpful and we are planning future work in this area. One obvious path is to replicate the capabilities of the TLA$^+$ model checker within Coq. We might even be able to prove the correctness of the model checker within Coq itself.

Further work could involve connecting TLA$^{\mathrm{Coq}}$ with other Coq based verification approaches, with the goal of "full stack verification," where low level implementations are proven to correspond to our high level models. One can imagine a world in which design, development, and specification happen in tandem, such that implementations are fully verified and correct by construction. Formalizing the high level descriptions of systems as we have done with TLA$^{\mathrm{Coq}}$ is an essential step in this effort.

Acknowledgement

Sandia National Laboratories is a multi-program laboratory managed and operated by Sandia Corporation, a wholly owned subsidiary of Lockheed Martin Corporation, for the U.S. Department of Energy's National Nuclear Security Administration (NNSA) under contract DE-AC04-94AL85000. This work was funded by NNSA's Advanced Simulation and Computing (ASC) Program.

References

1. Abadi, M., and Lamport, L. The existence of refinement mappings. *Theor. Comput. Sci. 82*, 2 (May 1991), 253–284.
2. Abadi, M., and Lamport, L. Conjoining specifications. *ACM Trans. Program. Lang. Syst. 17*, 3 (May 1995), 507–535.
3. Abadi, M., and Merz, S. On TLA as a logic. In *Proceedings of the NATO Advanced Study Institute on Deductive Program Design* (Secaucus, NJ, USA, 1996), Springer-Verlag New York, Inc., pp. 235–271.
4. Boldo, S., Jourdan, J.-H., Leroy, X., and Melquiond, G. A formally-verified C compiler supporting floating-point arithmetic. In *ARITH, 21st IEEE International Symposium on Computer Arithmetic* (2013), IEEE Computer Society Press, pp. 107–115.
5. Chaudhuri, K., Doligez, D., Lamport, L., and Merz, S. The TLA+ proof system: Building a heterogeneous verification platform. In *Proceedings of the 7th International Colloquium Conference on Theoretical Aspects of Computing* (Berlin, Heidelberg, 2010), ICTAC'10, Springer-Verlag, pp. 44–44.
6. Cohen, E., and Lamport, L. Reduction in TLA. In *CONCUR'98 Concurrency Theory*, vol. 1466 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 1998, pp. 317–331.
7. development team, T. C. *The Coq proof assistant reference manual*. LogiCal Project, 2004. Version 8.0.
8. Gonthier, G. Formal proof the four-color theorem. *Notices of the AMS 55*, 11 (Dec. 2008), 1382–1393. http://www.ams.org/notices/200811/tx081101382p.pdf.
9. Lamport, L. The temporal logic of actions. *ACM Trans. Program. Lang. Syst. 16*, 3 (May 1994), 872–923.
10. Lamport, L. Refinement in state-based formalisms. Tech. rep., DEC Systems Research Center, 1996.
11. Lamport, L. *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.
12. Pugh, W. The omega test: A fast and practical integer programming algorithm for dependence analysis. In *Proceedings of the 1991 ACM/IEEE Conference on Supercomputing* (New York, NY, USA, 1991), Supercomputing '91, ACM, pp. 4–13.
13. Ricketts, D., Malecha, G., Alvarez, M. M., Gowda, V., and Lerner, S. Towards verification of hybrid systems in a foundational proof assistant. In *13. ACM/IEEE International Conference on Formal Methods and Models for Codesign, MEMOCODE 2015, Austin, TX, USA, September 21-23, 2015* (2015), IEEE, pp. 248–257.
14. Svenningsson, J., and Axelsson, E. Combining deep and shallow embedding for edsl. In *Trends in Functional Programming: 13th International Symposium, TFP 2012, St. Andrews, UK, June 12-14, 2012, Revised Selected Papers* (2013), pp. 21–36.
15. Wan, H., He, A., You, Z., and Zhao, X. Formal proof of a machine closed theorem in Coq. *Journal of Applied Mathematics* (2014).
16. Wenzel, M. *The Isabelle/Isar Reference Manual*, 2012.
17. Yu, Y., Manolios, P., and Lamport, L. Model checking TLA+ specifications. In *Correct Hardware Design and Verification Methods* (1999), Springer-Verlag, pp. 54–66.

A Composition From Refinement

We have two models which both refine basic along functions of their state spaces. We want a fourth model which refines both of them, but in a way in which those refinements are compatible, in that either function from the combined model to basic is identical. Further, we want the best such combined model in a particular sense: any model which refines both command and memory in such a way that the two refinements yield the same refinement should refine our combined total model. This general picture should

$$\begin{tikzcd}
 & Basic \arrow[dl, "r_1"'] \arrow[dr, "r_2"] & \\
Command \arrow[dr, "\iota_1"] \arrow[ddr, "i_1"'] & & Memory \arrow[dl, "\iota_1"'] \arrow[ddl, "i_2"] \\
 & Total \arrow[d, dotted, "u"] & \\
 & c &
\end{tikzcd}$$

Fig. 16. Composition as Push-out

be familiar to those who have encountered category theory as it is what is called, in categorical language, a “push-out" (Figure 16). Indeed, more than 20 years ago, Goguen suggested push-outs as an abstract, formalism independent, definition for the combination of components [3]. And, Fiadeiro and Maibaum already demonstrated the use of push-outs for combining specifications in a temporal logic setting in the early 1990s [1]. Unfortunately, it seems this work has been largely neglected and not previously applied to TLA.

However, push-outs of refinements have a simple definition in TLA$^{\text{Coq}}$. We simply take the pull-back of the underlying state spaces in the underlying category of types or sets (that is, the push-out in the opposite category recognizing that functions go from more concrete to more abstract while we have written refinement arrows from abstract to concrete) and use the conjunction of the specifications along their specifications. That is, the state space of the total model should, intuitively, be the subtype of the product of state spaces of `Memory` and `Command` where the two refinement mappings yield the same value in the state space of basic. The combined specification is constructed then as the conjunction of the two refined specifications mapped along the projections from the combined state space.

As mentioned, we can construct the pushout of two TLA$^{\text{Coq}}$ refinements by way of the pullback of the underlying state spaces together with the conjunction of specification. The pullback of the state spaces would usually be the subset of the pairs of the values from the refined state spaces where those values map to the same element in the original unrefined model.

However, this intuitive definition does not quite work for us because Coq’s module system is generative rather than applicative and so when `Memory` and `Command` instantiate `Basic` they generate different types. To get around this, we must define a projection between them. First, we define the total model parametrically as a module functor and instantiate `Basic` and `Command`.

```
Module Total (Params : MemParams).
  Module MEMORY := Memory (Params).
  Module COMMAND := Command (MEMORY.FROMMEM).
```

Then we define an isomorphism between their underlying state spaces.

```
  Definition castSt1 (st : MEMORY.BASIC.St) :
      COMMAND.BASIC.St :=
    match st with
    | {| MEMORY.BASIC.ready := ready;
         MEMORY.BASIC.status := status;
         MEMORY.BASIC.memory := memory;
         MEMORY.BASIC.output := output |} =>
           {|
             COMMAND.BASIC.ready := ready;
             COMMAND.BASIC.status := status;
             COMMAND.BASIC.memory := memory;
             COMMAND.BASIC.output := output |}
    end.
```

This allows us to say when the two states are the same.

```
Definition sameSt (st1 : MEMORY.BASIC.St)
    (st2 : COMMAND.BASIC.St) : Prop :=
    castSt1 st1 = st2.
```

With this in place, we can define the full specification of the total model.

```
Inductive St : Type := Make_St : forall st1 st2,
    sameSt (MEMORY.refinement_mapping st1)
           (COMMAND.refinement_mapping st2)
    -> St.

Definition p1 (st : St) : MEMORY.St :=
    match st with
    | Make_St m _ _ => m
    end.

Definition p2 (st : St) : COMMAND.St :=
    match st with
    | Make_St _ m _ => m
    end.

Definition Spec : Expr St :=
   (map p1 MEMORY.Spec) `/\ (map p2 COMMAND.Spec).
```

B Details of Embedding

An essential notion in TLA is the idea of stuttering invariance. The idea is that "stuttering steps", those steps where the state does not change, are irrelevant. Therefore, two behaviors which differ only by the addition and removal of stuttering steps should validate the same formulae. However, formalizing this idea is a bit of a challenge. LamportÕs solution is to define a function $\#$ from behaviors to behaviors which eliminates any stuttering steps except for those at the end (when all remaining steps are stuttering steps) [4]. Then, two behaviors s_1 and s_2 are stuttering equivalent if $\#s_1 = \#s_2$.

Unfortunately, this definition does not work very well in a constructive setting such as Coq. The problem is that $\#$ function is not generally computable as equality of states is not necessarily decidable and, even when use a type of states where it is, we can not decide if all remaining steps in a behavior are stuttering steps as that would require examining an infinite number of states.

We thus use an alternative account of stuttering equivalence. Our idea is to observe that if two behaviors s_1 and s_2 are stuttering equivalent then they must have the same first state ($s_1 n = s_2 n$). Moreover, if the first step in s_1 is a stuttering step then $s_1@1$ is stuttering equivalent to s_2 while if it is not then there must be some n such that $s_1@1$ is stuttering equivalent $s_2@n$ and where all the steps in s_2 up to n are stuttering steps meaning all the states of s_2 before $s_2@n$ are equal. In either case there is some n such that $s_1@1$ is stuttering equivalent to $s_2@n$ and where for all k less than n, $s_2 0 = s_2 k$. And, because stuttering equivalence is symmetric, it works the other way also: there is some m such that $s_1@m$ is stuttering equivalent to $s_2@1$ and where for all k less than m, $s_1 0 = s_1 k$.

```
Definition stutter_relation {A : Type}
  (R : Behavior A -> Behavior A -> Prop) :=
  (forall s s', R s s' -> s 0 = s' 0) /\
  (forall s s', R s s' -> exists m, R (s @ 1) (s' @ m)
         /\ forall k, k < m -> s' 0 = s' k) /\
  (forall s s', R s s' -> exists m, R (s @ m) (s' @ 1)
         /\ forall k, k < m -> s 0 = s k).

Definition stutter_equiv {A : Type} (s s' : Behavior A) : Prop
  := exists R, R s s' /\ stutter_relation R.

Theorem stutter_equiv_eval : forall A (e : Expr A) s s',
     stutter_equiv s s' -> eval e s <-> eval e s'.
```

Fig. 17. Stuttering Equivalence as Largest Stuttering Relation

Further, it follows that not only is this a true fact about stuttering equivalent behaviors but it is a complete account as well. Given any two behaviors s_1 and s_2 which have the same first element and where there are n and m such that $s_1 1$ is stuttering equivalent to $s_2 n$ with $s_2 0 = s_2 k$ for all k less than n, and $s_1 m$ stuttering equivalent to $s_2 1$ with

$s_1 k = s_1 0$ for all k less than m, it is immediate that s_1 and s_2 are stuttering equivalent as well. Our idea then is to define stuttering equivalence co-inductively as the largest relation which satisfies this property. In particular, we define a stuttering relation as any relation on behaviors which satisfies these properties and then say two behaviors are stuttering equivalent if they are related by some stuttering relation (see Figure 17). We can then show that stuttering equivalence is indeed an equivalence relation and that it is a stuttering relation. Finally, the meta theorem that the truth value of a formula is always stuttering invariant.

However, stuttering invariance is not just an interesting theorem about our semantics, it also is a useful property in constructing proofs. In Figure 15 we give the core induction principle for proving that properties always hold given a specification which takes advantage of the fact that we do not need to consider stuttering steps because all formulae are stuttering invariant.

Our type of expressions is parameterized by a type of states. And, we would expect that different specification would use different state types we cannot necessarily simply use implication in modeling refinement. Instead, we may need to consider implication under some refinement mapping. To make this notion precise we have a function for

```
Fixpoint map (C A : Type) (f : C -> A) (e : Expr A) : Expr C :=
  match e with
    | E_Predicate P => E_Predicate (fun st => P (f st))
    | E_AlwaysAction g R =>
        E_AlwaysAction (fun st => g (f st))
        (fun st st' => R (f st) (f st'))
    | E_EventuallyAction g R =>
        E_EventuallyAction (fun st => g (f st))
        (fun st st' => R (f st) (f st'))
    | `~ P => `~ (map f P)
    | P `/\ Q => (map f P) `/\ (map f Q)
    | P `\/ Q => (map f P) `\/ (map f Q)
    | P `=> Q => (map f P) `=> (map f Q)
    | P `<=> Q => (map f P) `<=> (map f Q)
    | [] P => [] (map f P)
    | <> P => <> (map f P)
    | E_ForallRigid g => E_ForallRigid (fun x => map f (g x))
    | E_ExistsRigid g => E_ExistsRigid (fun x => map f (g x))
  end.
```

Fig. 18. Mapping over an expression to change the state type

mapping a state change function over a formula (given in Figure 18). This mapping procedure satisfies the important property which is that mapping over formula is equivalent to mapping over behaviors.

```
Lemma map_id : forall (A : Type) (e : Expr A),
  map (fun x => x) e = e.
```

As the mathematical language in our embedding is Coq instead of ZFC, our system differs somewhat from TLA+. In particular, Coq is always typed and so the state of formulae we work with be given as some type. This differs from TLA+ where the state is always a mapping which assigns each (of countably many) variables an arbitrary set. However, from the type theoretic perspective these TLA+ states can be given a type: namely the type of functions from the type of variables to the type of sets. We can therefore represent TLA+ specification directly without assigning types to the variables if we can simply exhibit a type in Coq which behaves like the collection of all sets in set theory. As a proof of concept of this idea we have constructed such a type using a variant of the canonical model of constructive set theory in type theory as well founded trees indexed by a universe due to Martin-Löf and Aczel [2]. A full version of ZFC can be then derived by assuming additional axioms for choice and classicality. However, our experience has been that working with typed representations of states works better in practice than encoding everything into sets. Indeed, it seems that variables in actual TLA models are almost always “typed" in the sense that their values do not range over arbitrary sets but rather some more restricted collection of values such as numbers or lists. Furthermore, these types are an inherent part of the specification known to designers from the beginning and it simplifies things significantly if they hold definitionally rather than as a theorem which must be shown. However, it may make sense to at some future time to parse models written for the TLA+ toolset and then automatically produce TLA$^{\text{Coq}}$ specifications from these models. Using the type of sets may be helpful in this situation.

References

1. FIADEIRO, J., AND MAIBAUM, T. Temporal theories as modularisation units for concurrent system specification. *Formal Aspects of Computing 4*, 3 (1992), 239–272.
2. GAMBINO, N., AND ACZEL, P. The generalised type-theoretic interpretation of constructive set theory. *Journal of Symbolic Logic 71* (3 2006), 67–103.
3. GOGUEN, J. A. A categorical manifesto. *Mathematical structures in computer science 1*, 01 (1991), 49–67.
4. LAMPORT, L. The temporal logic of actions. *ACM Trans. Program. Lang. Syst. 16*, 3 (May 1994), 872–923.

C Implementation of the TLA$^{\text{Coq}}$

C.1 Behavior.v

```
Require Import Arith.
Require Import FunctionalExtensionality.
Require Import Program.
Require Import Omega.

Set Implicit Arguments.

Definition Behavior (A : Type) := nat → A.

Section shift.
```

```
  Variable A : Type.
  Implicit Types s : Behavior A.
  Definition shift s n : Behavior A :=
    fun t ⇒ s (t + n).
  Lemma unshift : ∀ s n i, shift s n i = s (i+n).
  Proof. unfold shift. auto. Qed.
  Lemma shift_0 : ∀ s, shift s 0 = s.
  Proof.
    intros s; extensionality x; replace (s x) with (s (x + 0)) by auto;
apply unshift.
  Qed.
  Lemma contraction : ∀ s n m, shift (shift s n) m = shift s (m + n).
  Proof.
    unfold shift; intros s n m; extensionality x; rewrite plus_assoc;
auto.
  Qed.
End shift.
Delimit Scope behavior_scope with behavior.
Notation "s @ n" :=
  (shift s n)
    (at level 1, no associativity) : behavior_scope.
Definition stutter_relation {A : Type} (R : Behavior A → Behavior A → Prop)
:=
  (∀ s s', R s s' → s 0 = s' 0) ∧
  (∀ s s', R s s' → ∃ m, R (shift s 1) (shift s' m) ∧ ∀ k, k < m → s' 0 = s' k) ∧
  (∀ s s', R s s' → ∃ m, R (shift s m) (shift s' 1) ∧ ∀ k, k < m → s 0 = s k).
Definition stutter_equiv {A : Type} (s s' : Behavior A) : Prop :=
  ∃ R, R s s' ∧ stutter_relation R.
Lemma stutter_equiv_stuttering_relation : ∀ A, stutter_relation (stutter_equiv : Behav-
ior A → Behavior A → Prop).
Proof.
  intros A.
  repeat split; intros s s' H; destruct H as (R,(H,(H0,(H1,H2)))); auto.
  apply H1 in H. destruct H as (m,(H,H3)); ∃ m; split; auto; ∃ R; repeat
split; auto.
  apply H2 in H. destruct H as (m,(H,H3)); ∃ m; split; auto; ∃ R; repeat
split; auto.
Qed.
Lemma stutter_relation_univ : ∀ A (R : Behavior A → Behavior A → Prop),
                              stutter_relation R →
                              ∀ s s',
                                R s s' →
```

```
                             ∀ n, ∃ m, R (shift s n) (shift s' m).
Proof.
  intros A R H s s' H0 n.
  generalize dependent s. generalize dependent s'.
  induction n.
  intros s' s H0. ∃ 0. do 2 rewrite shift_0. auto.
  intros s' s H0. destruct H as (H,(H1,H2)).
  apply H1 in H0. destruct H0 as (m,(H0,H3)).
  apply IHn in H0. destruct H0 as (m0,H0).
  do 2 rewrite contraction in H0.
  ∃ (m0 + m). replace (S n) with (n + 1) by omega. auto.
Qed.

Theorem stutter_equiv_univ : ∀ A (s s' : Behavior A), stutter_equiv s s' →
                                                                ∀ n, ∃ m,
                                                                  stutter_equiv
(shift s n) (shift s' m).
Proof.
  intros A s s' H n.
  apply stutter_relation_univ; trivial.
  apply stutter_equiv_stuttering_relation.
Qed.

Lemma id_stutter_prop : ∀ A, stutter_relation (fun (s s': Behavior A) ⇒ s = s').
Proof.
  intros A.
  repeat split; intros s s' H; rewrite H; auto; ∃ 1; split; auto;
  intros k H0; replace k with 0 by omega; auto.
Qed.

Theorem refl_stutter_equiv : ∀ A (s : Behavior A), stutter_equiv s s.
Proof.
  intros A s.
  ∃ (fun s s' ⇒ s = s'). split; trivial.
  apply id_stutter_prop.
Qed.

Lemma flip_stutter_relation : ∀ A (R : Behavior A → Behavior A → Prop),
                              stutter_relation R →
                              stutter_relation (fun s s' ⇒ R s' s).
Proof.
  intros A R H.
  destruct H as (H,(H0,H1)).
  repeat split; auto.
  intros s s' H2. symmetry. auto.
Qed.

Theorem sym_stutter_equiv : ∀ A (s s' : Behavior A),
                               stutter_equiv s s'
                              → stutter_equiv s' s.
Proof.
  intros A s s' H.
  destruct H as (R,(H,RS)).
  ∃ (fun s s' ⇒ R s' s).
  split; trivial.
  apply flip_stutter_relation; trivial.
Qed.

Lemma le_difference : ∀ n m, n ≤ m → ∃ r, m = n + r.
Proof.
  intros n m H.
  generalize dependent n.
  induction m.
  intros n H. inversion H. ∃ 0; omega.
  intros n H.
  inversion H.
  ∃ 0; omega.
  assert (T : ∃ r, m = n + r). apply IHm; auto.
  destruct T as (r, T). ∃ (S r). omega.
Qed.

Lemma stutter_relation_univ_small : ∀ A R (s s' : Behavior A) n,
                                        stutter_relation R →
                                        R s s' →
                                        (∀ k, k < n → s 0 = s k) →
                                        ∃ m, R (shift s n) (shift s' m) ∧
                                                  (∀ k, k < m → s' 0 = s' k).
Proof.
  intros A R s s' n H H0 H1.
  generalize dependent s. generalize dependent s'.
  induction n.
  intros s' s H0 H1. ∃ 0. split. do 2 rewrite shift_0. auto.
  intros k H2. exfalso. omega.
  intros s' s H0 H1.
  destruct H as (R_zero,(R_left,R_right)).
  assert (H2 : ∃ m, R (shift s n) (shift s' m) ∧ (∀ k, k < m → s' 0 = s' k)).
  apply IHn; auto.
  destruct H2 as (m,(H2,H3)).
  assert (H4 : ∃ m0,
                   R (shift (shift s n) 1) (shift (shift s' m) m0)
                   ∧ ∀ k, k < m0 → (shift s' m) 0 = (shift s' m) k).
  apply R_left; auto.
  destruct H4 as (m0,(H4,H5)).
  unfold shift in H5. simpl in H5.
  assert (H6 : s 0 = (shift s' m) 0). transitivity ((shift s n) 0). unfold shift.
simpl. apply H1.
```

```
  omega. apply R_zero; auto.
  unfold shift in H6. simpl in H6.
  ∃ (m0 + m). split.
  do 2 rewrite contraction in H4. simpl in H4. auto.
  intros k H.
  assert (H7 : k < m ∨ m ≤ k). omega.
  destruct H7 as [H7|H7]. apply H3; auto.
  apply le_difference in H7.
  destruct H7 as (r,H7).
  assert (H8 : k = r + m). omega. clear H7.
  rewrite H8 in *.
  transitivity (s' m). transitivity (s 0); auto. symmetry. apply R_zero;
auto.
  apply H5. omega.
Qed.

Lemma comp_stutter_prop : ∀ A (R1 R2 : Behavior A → Behavior A → Prop),
                                 stutter_relation R1 →
                                 stutter_relation R2 →
                                 stutter_relation (fun s s' ⇒ ∃ s'', R1 s s'' ∧ R2 s''
s').
Proof.
  intros A R1 R2 H H0.
  repeat split; intros s s' H1; destruct H1 as (t,(H1,H2)).
  destruct H as (H,_). destruct H0 as (H0,_). transitivity (t 0); auto.
  destruct H as (R1_zero,(R1_right,R1_left)).
  apply R1_right in H1. destruct H1 as (m,(H1,H3)).
  assert (H4 : ∃ m0, R2 (shift t m) (shift s' m0) ∧ ∀ k, k < m0 → s' 0 = s' k).
  apply stutter_relation_univ_small; auto.
  destruct H4 as (m0,(H4,H5)). ∃ m0. split; auto. ∃ (shift t m). split; auto.
  destruct H0 as (R2_zero,(R2_right,R2_left)).
  assert (H3 : ∃ m : nat,
              R2 (shift t m) (shift s' 1) ∧
              (∀ k : nat, k < m → t 0 = t k)). apply R2_left; auto.
  destruct H3 as (m,(H3,H4)).
  set (R1_flip := fun a b ⇒ R1 b a).
  assert (H5 : ∃ m0, R1_flip (shift t m) (shift s m0) ∧
                          (∀ k, k < m0 → s 0 = s k)).
  apply stutter_relation_univ_small; auto.
  unfold R1_flip.
  apply flip_stutter_relation. auto.
  unfold R1_flip in *. clear R1_flip.
  destruct H5 as (m0,(H5,H6)).
  ∃ m0. split; auto. ∃ (shift t m); split; auto.
Qed.

Theorem trans_stutter_equiv : ∀ A (s t s' : Behavior A),
                                                stutter_equiv s t →
                                                stutter_equiv t s' →
                                                stutter_equiv s s'.
Proof.
  intros A s t s' H H0.
  destruct H as (R1,(H,R1_rel)).
  destruct H0 as (R2,(H0,R2_rel)).
  ∃ (fun a b ⇒ ∃ c, R1 a c ∧ R2 c b).
  split. ∃ t. split; auto.
  apply comp_stutter_prop; auto.
Qed.

Definition prop_extension {A} (R : Behavior A → Behavior A → Prop) :=
  fun s s' ⇒ R s s' ∨ (s 0 = s' 0 ∧ R (shift s 1) (shift s' 1)).

Lemma extend_stutter_relation : ∀ A (R : Behavior A → Behavior A → Prop),
                                                  stutter_relation R →
                                                  stutter_relation (prop_extension R).
Proof.
  intros A R H.
  destruct H as (H,(H0,H1)).
  repeat split; intros s s' H2; destruct H2 as [H2|(H2,H3)]; auto.
  apply H0 in H2. destruct H2 as (m,(H2,H3)). ∃ m. split; auto. left;
auto.
  ∃ 1. split. left. auto. intros k H4. replace k with 0 by omega; auto.
  apply H1 in H2. destruct H2 as (m,(H2,H3)). ∃ m. split; auto. left;
auto.
  ∃ 1. split. left. auto. intros k H4. replace k with 0 by omega; auto.
Qed.

Theorem extend_stutter_equiv : ∀ A (s s' : Behavior A), s 0 = s' 0 →
                                                          stutter_equiv (shift s 1) (shift s'
1) →
                                                          stutter_equiv s s'.
Proof.
  intros A s s' H H0.
  destruct H0 as (R,(H0,H1)).
  ∃ (prop_extension R). split.
  right; split; auto.
  apply extend_stutter_relation; auto.
Qed.

Definition front_relation {A} (s s' : Behavior A) := ∃ m n, s 0 = s' 0 ∧
                                                                          (∀ k, k
≤ m → s 0 = s k) ∧
                                                                          (∀ k, k
≤ n → s' 0 = s' k) ∧
```

```
                                                                    shift s
m = shift s' n.

Lemma front_relation_refl : ∀ A (s : Behavior A), front_relation s s.
Proof.
  intros A s.
  ∃ 0. ∃ 0. repeat split; auto; intros k H; replace k with 0 by omega;
auto.
Qed.

Lemma front_relation_step : ∀ A (s s' : Behavior A), front_relation s s' →
    ∃ m : nat,
      front_relation s @ 1%behavior s' @ m%behavior ∧
      (∀ k : nat, k < m → s' 0 = s' k).
Proof.
  intros A s s' H.
  destruct H as (m,(n,(same_zero,(lt_m,(lt_n,same_tail))))).
  destruct m. ∃ (S n). split. rewrite shift_0 in same_tail.
  rewrite same_tail. rewrite contraction. simpl. apply front_relation_refl.
  intros k H. apply lt_n; omega.
  ∃ n. split. ∃ m. ∃ 0.
  assert (same_one : s 0 = s 1). apply lt_m; omega.
  repeat split.
  unfold shift. simpl. transitivity (s 0). auto.
  transitivity (s' 0); auto.
  intros k H. unfold shift. simpl. transitivity (s 0); auto.
  apply lt_m; omega.
  intros k H. replace k with 0 by omega. auto.
  do 2 rewrite contraction. simpl. replace (m + 1) with (S m) by omega.
auto.
  intros k H. apply lt_n; omega.
Qed.

Lemma front_relation_sym : ∀ A (s s' : Behavior A), front_relation s s' →
                                                          front_relation s'
s.
Proof.
  intros A s s' H.
  destruct H as (m,(n,(same_refl,(lt_m,(lt_n,same_tail))))).
  ∃ n. ∃ m. repeat split; auto.
Qed.

Lemma front_stutter_relation : ∀ A, stutter_relation (front_relation : Behavior A →
Behavior A → Prop).
Proof.
  intros A.
  repeat split; intros s s' H.
  destruct H as (m,(n,(same_zero,(lt_m,(lt_n,same_tail))))); auto.
  apply front_relation_step; auto.
  apply front_relation_sym in H. apply front_relation_step in H.
  destruct H as (m,(H,H0)).
  ∃ m. split; auto. apply front_relation_sym; auto.
Qed.
Lemma front_relation_shift_1 : ∀ A (s : Behavior A), s 0 = s 1 →
                                                        front_relation (shift
s 1) s.
Proof.
  intros A s H.
  ∃ 0. ∃ 1. repeat split; auto.
  intros k H0. replace k with 0 by omega; auto.
  intros k H0. assert (H1 : k = 0 ∨ k = 1). omega. destruct H1 as [H1|H1];
rewrite H1; auto.
  rewrite contraction. auto.
Qed.
Theorem stutter_equiv_shift_1 : ∀ A (s : Behavior A), s 0 = s 1 →
                                                        stutter_equiv (shift
s 1) s.
Proof.
  intros A s H.
  ∃ front_relation. split.
  apply front_relation_shift_1; auto.
  apply front_stutter_relation; auto.
Qed.
Theorem stutter_equiv_induct_next : ∀ A (s s' : Behavior A) R,
                                  stutter_equiv s s' →
                                  (∀ n, R (s n) (s (S n)) ∨ s n = s (S n)) →
                                  (∀ n, R (s' n) (s' (S n)) ∨ s' n = s' (S n)).
Proof.
  intros A s s' R H H0 n.
  assert (H1 : ∃ m, stutter_equiv (shift s' n) (shift s m)).
  apply stutter_equiv_univ. apply sym_stutter_equiv; auto.
  destruct H1 as (m,H1).
  assert (H2 : stutter_relation (stutter_equiv : Behavior A → Behavior A → Prop)).
  apply stutter_equiv_stuttering_relation.
  destruct H2 as (same_zero,(step_left,step_right)).
  assert (H2 : (shift s' n) 0 = (shift s m) 0). apply same_zero. auto.
  unfold shift in H2. simpl in H2.
  rewrite H2.
  apply step_left in H1.
  destruct H1 as (m0,(H1,H3)).
  destruct m0.
  assert (H4 : (shift (shift s' n) 1) 0 = (shift (shift s m) 0) 0). apply same_zero;
auto.
```

```
  unfold shift in H4. simpl in H4.
  rewrite H4. right; auto.
  assert (H4 : (shift s m) 0 = (shift s m) m0). apply H3; omega.
  unfold shift in H4. simpl in H4.
  assert (H5 : (shift (shift s' n) 1) 0 = (shift (shift s m) (S m0)) 0). apply same_zero;
auto.
  unfold shift in H5. simpl in H5.
  rewrite H4. rewrite H5.
  apply H0.
Qed.

Theorem stutter_equiv_same_first : ∀ A (s s' : Behavior A),
                                    stutter_equiv s s' →
                                    s 0 = s' 0.
Proof.
  intros A s s' H.
  destruct H as (r,(H,(same_zero,_))).
  auto.
Qed.

Theorem stutter_equiv_induct_next'_base
    : ∀ (A : Type) (s s' : Behavior A) (R : A → A → Prop),
      stutter_equiv s s' →
      R (s 0) (s 1) → s 0 ≠ s 1 →
      ∃ n : nat, R (s' n) (s' (S n)).
Proof.
  intros A s s' R H H1 H2.
  pose (H3 := stutter_equiv_stuttering_relation A).
  destruct H3 as (H3,(H4,H5)).
  assert (H6 : ∃ m, stutter_equiv s @ 1%behavior s' @ m%behavior ∧
      (∀ k : nat, k < m → s' 0 = s' k)). apply H4; auto.
  destruct H6 as (m,(H6,H7)).
  assert (H0 : s 0 = s' 0). apply H3; auto.
  destruct m.
  apply H3 in H6. unfold shift in H6. simpl in H6.
  rewrite H0 in H2. rewrite H6 in H2. exfalso. apply H2; auto.
  ∃ m.
  apply H3 in H6. unfold shift in H6. simpl in H6.
  rewrite ← H6.
  specialize H7 with m. rewrite ← H7; auto.
  rewrite ← H0.
  auto.
Qed.

Theorem stutter_equiv_induct_next'
    : ∀ (A : Type) (s s' : Behavior A) (R : A → A → Prop),
      stutter_equiv s s' →
      (∃ n : nat, R (s n) (s (S n)) ∧ s n ≠ s (S n)) →
      ∃ n : nat, R (s' n) (s' (S n)).
Proof.
  intros A s s' R H H1.
  destruct H1 as (n,(H1,H2)).
  assert (H3 : ∃ m, stutter_equiv (shift s n) (shift s' m)). apply stutter_equiv_univ;
auto.
  destruct H3 as (m,H3).
  assert (H4 : ∃ n0, R ((s' @ m%behavior) n0) ((s' @ m%behavior) (S n0))).
  apply stutter_equiv_induct_next'_base with (s @ n%behavior); auto.
  destruct H4 as (n0,H4).
  unfold shift in H4. simpl in H4.
  ∃ (n0 + m); auto.
Qed.

Theorem stutter_equiv_induct_next''
      : ∀ (A : Type) (s s' : Behavior A) (R : A → A → Prop) (B : Type) (f : A → B),
        stutter_equiv s s' →
        (∃ n : nat, R (s n) (s (S n)) ∧ f (s n) ≠ f (s (S n))) →
        ∃ n : nat, R (s' n) (s' (S n)) ∧ f (s' n) ≠ f (s' (S n)).
Proof.
  intros A s s' R B f H H1.
  destruct H1 as (n,(H1,H2)).
  set (M := fun x y ⇒ R x y ∧ f x ≠ f y).
  assert (H3 : ∃ m, M (s' m) (s' (S m))).
  apply stutter_equiv_induct_next' with s; auto.
  ∃ n. unfold M. repeat split; auto. intros N. apply H2; rewrite N;
auto.
  auto.
Qed.

Require Import Relations.
Require Import Setoid.
Require Import Morphisms.
Import RelationClasses.

Add Parametric Relation A : (Behavior A) (@stutter_equiv A)
    reflexivity proved by (fun s ⇒ refl_stutter_equiv s)
    symmetry proved by (fun s s' H ⇒ sym_stutter_equiv H)
    transitivity proved by (fun s t s' H0 H1 ⇒ trans_stutter_equiv H0 H1)
      as stutter_equiv_rel.
```

C.2 Expr.v

```
Set Implicit Arguments.

Require Import Logic.FunctionalExtensionality.
Require Import Behavior.
Local Open Scope behavior.
```

```
Section Syntax.
  Variable A : Type.
  Definition Predicate := A → Prop.
  Definition Action := A → A → Prop.
  Inductive Expr :=
  | E_Predicate : Predicate → Expr
  | E_AlwaysAction : ∀ {B : Type}, (A → B) → Action → Expr
  | E_EventuallyAction : ∀ {B : Type}, (A → B) → Action → Expr
  | E_Neg : Expr → Expr
  | E_Conj : Expr → Expr → Expr
  | E_Disj : Expr → Expr → Expr
  | E_Impl : Expr → Expr → Expr
  | E_Equiv : Expr → Expr → Expr
  | E_Always : Expr → Expr
  | E_Eventually : Expr → Expr
  | E_ForallRigid : ∀ {B : Type}, (B → Expr) → Expr
  | E_ExistsRigid : ∀ {B : Type}, (B → Expr) → Expr.
  Definition E_Enabled (r : Action) :=
    E_Predicate (fun st ⇒ ∃ st', r st st').
End Syntax.

Delimit Scope tla_scope with tla.
Notation "P `<=> Q" :=
  (E_Equiv P Q) (at level 95, no associativity) : tla_scope.
Notation "P `=> Q" :=
  (E_Impl P Q) (at level 90, right associativity) : tla_scope.
Notation "P `\/ Q" :=
  (E_Disj P Q) (at level 85, right associativity) : tla_scope.
Notation "P `/\ Q" :=
  (E_Conj P Q) (at level 80, right associativity): tla_scope.
Notation "`~ P" :=
  (E_Neg P) (at level 75, right associativity) : tla_scope.
Notation "[] P" :=
  (E_Always P) (at level 75, right associativity) : tla_scope.
Notation "<> P" :=
  (E_Eventually P) (at level 75, right associativity) : tla_scope.
Definition E_LeadsTo {T} (P Q : Expr T) := E_Always (E_Impl P (E_Eventually
Q)).
Notation "P `~> Q" :=
  (E_LeadsTo P Q) (at level 90, right associativity) : tla_scope.
Notation "'ENABLED' r" :=
  (E_Enabled r) (at level 70, no associativity) : tla_scope.
Notation "[][ R ]" := (E_AlwaysAction (fun x ⇒ x) R) : tla_scope.
Notation "<>< R >" := (E_EventuallyAction (fun x ⇒ x) R) : tla_scope.

Notation "‘ P" :=
  (E_Predicate P) (at level 0, right associativity) : tla_scope.
Notation "’FORALL’ p" := (E_ForallRigid p) (at level 70, right associativity)
                         : tla_scope.
Notation "’EXISTS’ p" := (E_ForallRigid p) (at level 70, right associativity)
                         : tla_scope.

Local Open Scope tla.
Definition WF {T} (A : Action T) : Expr T :=
  ([] (E_EventuallyAction (fun x ⇒ x) A)) ‘\/ ([]<>(‘~ ENABLED A)).
Definition SF {T} (A : Action T) : Expr T :=
  ([] (E_EventuallyAction (fun x ⇒ x) A)) ‘\/ (<>[](‘~ ENABLED A)).
Arguments WF _ A%tla.
Arguments SF _ A%tla.
Fixpoint eval (A : Type) (e : Expr A) : Behavior A → Prop :=
  match e with
    | E_Predicate p ⇒ fun s ⇒ p (s 0)
    | E_AlwaysAction f r ⇒ fun s ⇒ ∀ n, (r (s@n 0) (s@n 1)) ∨ f (s@n 0) = f (s@n
1)
    | E_EventuallyAction f r ⇒ fun s ⇒ ∃ n, (r (s@n 0) (s@ n 1)) ∧ f (s@n 0) ≠ f
(s@n 1)
    | ‘~ e ⇒ fun s ⇒ ¬ eval e s
    | e1 ‘/\ e2 ⇒ fun s ⇒ eval e1 s ∧ eval e2 s
    | e1 ‘\/ e2 ⇒ fun s ⇒ eval e1 s ∨ eval e2 s
    | e1 ‘=> e2 ⇒ fun s ⇒ eval e1 s → eval e2 s
    | e1 ‘<=> e2 ⇒ fun s ⇒ eval e1 s ↔ eval e2 s
    | [] e ⇒ fun s ⇒ ∀ n, eval e (s@n)
    | ≠ e ⇒ fun s ⇒ ∃ n, eval e (s@n)
    | E_ForallRigid f ⇒ fun s ⇒ ∀ x, eval (f x) s
    | E_ExistsRigid f ⇒ fun s ⇒ ∃ x, eval (f x) s
  end.
Definition valid (A : Type) (e : Expr A) : Prop :=
  ∀ s, eval e s.
Arguments valid _ e%tla.
Arguments eval _ e%tla s%behavior.
Fixpoint map (C A : Type) (f : C → A) (e : Expr A) : Expr C :=
  match e with
    | E_Predicate P ⇒ E_Predicate (fun st ⇒ P (f st))
    | E_AlwaysAction g R ⇒ E_AlwaysAction (fun st ⇒ g (f st)) (fun st st' ⇒ R (f st) (f st'))
    | E_EventuallyAction g R ⇒ E_EventuallyAction (fun st ⇒ g (f st)) (fun st st' ⇒ R (f st) (f st'))
    | `~ P ⇒ `~ (map f P)
    | P `/\ Q ⇒ (map f P) `/\ (map f Q)
    | P `\/ Q ⇒ (map f P) `\/ (map f Q)
    | P `=> Q ⇒ (map f P) `=> (map f Q)
    | P `<=> Q ⇒ (map f P) `<=> (map f Q)
    | [] P ⇒ [] (map f P)
    | ≠ P ⇒ ≠ (map f P)
    | E_ForallRigid g ⇒ E_ForallRigid (fun x ⇒ map f (g x))
    | E_ExistsRigid g ⇒ E_ExistsRigid (fun x ⇒ map f (g x))
  end.

Lemma same_args_same_res : ∀ A, ∀ B, ∀ (f : A → B), ∀ x y, x = y → f x = f y.
    intros A B f x y H; rewrite H; auto.
Qed.

Lemma map_id :
  ∀ (A : Type) (e : Expr A),
    map (fun x ⇒ x) e = e.
Proof.
  assert (arg : ∀ A, ∀ B, ∀ (f : A → B), ∀ x y, x = y → f x = f y). intros A B f x y H; rewrite H; auto.
  induction e; simpl;
  try (rewrite IHe1; rewrite IHe2; trivial);
  try (rewrite IHe; trivial);
  try trivial;
  try (apply same_args_same_res; extensionality x; auto).
Qed.

Lemma map_map :
  ∀ (A B C : Type) (f : A → B) (g : B → C) (e : Expr C),
    map f (map g e) = map (fun x ⇒ g (f x)) e.
Proof.
  induction e; simpl;
  try (rewrite IHe1; rewrite IHe2; trivial);
  try (rewrite IHe; trivial);
  try trivial;
  try (apply same_args_same_res; extensionality x; auto).
Qed.

Lemma slide_shift : ∀ A B (f : A → B) s n, (fun x ⇒ f (s x)) @ n = fun x ⇒ f ((s @ n) x).
Proof.
  intros A B f s n.
  extensionality x.
  induction n; auto.
Qed.

Lemma map_eval :
  ∀ (A B : Type) (f : A → B) (e : Expr B) (s : Behavior A),
    eval (map f e) s ↔ eval e (fun x ⇒ f (s x)).
Proof.
  intros A B f e s. generalize dependent s.
  induction e; simpl; intros;
  try (rewrite IHe1; rewrite IHe2; solve [intuition]);
  try (rewrite IHe; solve [intuition]);
  try solve [intuition].
  split; intros H n;
  [ try (rewrite slide_shift; apply IHe; trivial) |
    specialize H with n; rewrite slide_shift in H; apply IHe in H; trivial].
  split; intros H; destruct H as (n,H); ∃ n;
  [ rewrite slide_shift; apply IHe; trivial |
    rewrite slide_shift in H; apply IHe in H; trivial].
  split; intros H0 x; apply H; trivial.
  split; intros H0; destruct H0 as (x,H0); ∃ x; apply H; trivial.
Qed.

Theorem stutter_equiv_eval : ∀ A (e : Expr A) (s s' : Behavior A), stutter_equiv s s'
→
                                                                              eval
e s ↔
                                                                              eval
e s'.
Proof.
  intros A e.
  induction e; intros s s' H0; simpl.
  apply stutter_equiv_same_first in H0; rewrite H0; split; auto.
  split; intros H1 n; unfold shift in *; simpl in *;
  set (a' := fun x y ⇒ a x y ∨ b x = b y).
  assert (H2 : a' (s' n) (s' (S n)) ∨ s' n = s' (S n)).
  apply stutter_equiv_induct_next with s; auto. intros n0.
  specialize H1 with n0. destruct H1 as [H1|H1]; left. left; auto.
right; auto.
  destruct H2 as [[H2|H2]|H2]. left; auto. right; auto. rewrite H2;
right; auto.
  assert (H2 : a' (s n) (s (S n)) ∨ s n = s (S n)).
  apply stutter_equiv_induct_next with s'; auto. symmetry; auto. intros n0.
  specialize H1 with n0. destruct H1 as [H1|H1]; left. left; auto.
right; auto.
  destruct H2 as [[H2|H2]|H2]. left; auto. right; auto. rewrite H2;
right; auto.
  split; intros H1.
  apply stutter_equiv_induct_next'' with s; auto.
  apply stutter_equiv_induct_next'' with s'; auto.
  assert (H : s = fun n ⇒ s @ n 0). extensionality n; auto.
  rewrite ← H. symmetry. auto.
  split; intros H1 H2; apply H1. apply IHe with s'; auto. apply IHe
with s; auto. symmetry; auto.
  assert (H3 : stutter_equiv s s'); auto.
  apply IHe1 in H0. apply IHe2 in H3.
  intuition.
  assert (H3 : stutter_equiv s s'); auto.
  apply IHe1 in H0. apply IHe2 in H3.
  split; intros H1; destruct H1; intuition.
  assert (H3 : stutter_equiv s s'); auto.
  apply IHe1 in H0. apply IHe2 in H3.
  intuition.
  assert (H3 : stutter_equiv s s'); auto.
  apply IHe1 in H0. apply IHe2 in H3.
  intuition.
  split; intros H1 n.
  assert (H2: ∃ m, stutter_equiv (s' @ n) (s @ m)). apply stutter_equiv_univ.
symmetry; auto.
  destruct H2 as (m,H2). apply IHe in H2. intuition.
  assert (H2 : ∃ m, stutter_equiv (s @ n) (s' @ m)). apply stutter_equiv_univ;
auto.
  destruct H2 as (m,H2). apply IHe in H2. intuition.
  split; intros H1; destruct H1 as (n,H1).
  assert (H2 : ∃ m, stutter_equiv (s @ n) (s' @ m)). apply stutter_equiv_univ;
auto.
  destruct H2 as (m,H2). apply IHe in H2. ∃ m; intuition.
  assert (H2 : ∃ m, stutter_equiv (s' @ n) (s @ m)). apply stutter_equiv_univ.
symmetry; auto.
  destruct H2 as (m,H2). apply IHe in H2. ∃ m; intuition.
  split; intros H1 x; specialize H with x s s'; intuition.
  split; intros H1; destruct H1 as (x,H1); specialize H with x s s'; ∃
x; intuition.
Qed.
```

C.3 ExprEquiv.v

```
Require Import Behavior.
Require Import Expr.

Definition expr_equiv {A : Type} (s : Behavior A) (P Q : Expr A) : Prop :=
  eval (P `<=> Q) s.

Require Import Relations.
Require Import Setoid.
Require Import Morphisms.
Import RelationClasses.

Lemma expr_equiv_refl :
  ∀ A s, reflexive _ (@expr_equiv A s).
Proof. firstorder. Qed.

Lemma expr_equiv_sym :
  ∀ A s, symmetric _ (@expr_equiv A s).
Proof. firstorder. Qed.

Lemma expr_equiv_trans :
  ∀ A s, transitive _ (@expr_equiv A s).
Proof. firstorder. Qed.

Add Parametric Relation A s : (Expr A) (@expr_equiv A s)
    reflexivity proved by (expr_equiv_refl A s)
    symmetry proved by (expr_equiv_sym A s)
    transitivity proved by (expr_equiv_trans A s)
      as expr_equiv_rel.

Add Parametric Morphism A s : (fun e ⇒ @eval A e s) with
    signature @expr_equiv A s ==> iff as expr_eval_mor.
Proof. firstorder. Qed.

Add Parametric Morphism A s : (@E_Neg A) with
    signature @expr_equiv A s ==> @expr_equiv A s as expr_neg_mor.
Proof. firstorder. Qed.

Add Parametric Morphism A s : (@E_Conj A) with
    signature @expr_equiv A s ==> @expr_equiv A s ==> @expr_equiv A s as expr_conj_mor.
Proof. firstorder. Qed.

Add Parametric Morphism A s : (@E_Disj A) with
    signature @expr_equiv A s ==> @expr_equiv A s ==> @expr_equiv A s as expr_disj_mor.
Proof. firstorder. Qed.

Add Parametric Morphism A s : (@E_Impl A) with
    signature @expr_equiv A s ==> @expr_equiv A s ==> @expr_equiv A s as expr_impl_mor.
Proof. firstorder. Qed.

Add Parametric Morphism A s : (@E_Equiv A) with
    signature @expr_equiv A s ==> @expr_equiv A s ==> @expr_equiv A s as expr_equiv_mor.
Proof. firstorder. Qed.

Add Parametric Morphism (A B : Set) s (f : A → B) : (@map A B f) with
    signature @expr_equiv B (fun i ⇒ f (s i)) ==> @expr_equiv A s as expr_map_mor.
Proof.
  unfold expr_equiv, valid. simpl. intros e1 e2 H.
  do 2 try (rewrite map_eval).
  assumption.
Qed.

Section simpl_map.
```

```
  Local Open Scope tla.

  Variables A B : Type.
  Variable f : A → B.

  Lemma simpl_map_always :
    ∀ s e, eval (map f ([] e)) s ↔ eval ([] (map f e)) s.
  Proof.
    reflexivity.
  Qed.

  Lemma simpl_map_conj :
    ∀ s e1 e2, eval (map f (e1 `/\ e2)) s ↔ eval (map f e1 `/\ map f e2) s.
  Proof.
    reflexivity.
  Qed.

  Lemma simpl_map_disj :
    ∀ s e1 e2, eval (map f (e1 `\/ e2)) s ↔ eval (map f e1 `\/ map f e2) s.
  Proof.
    reflexivity.
  Qed.

End simpl_map.

Ltac simpl_map :=
  repeat (rewrite simpl_map_always || rewrite simpl_map_conj || rewrite
simpl_map_disj).
```

C.4 Invariant.v

```
Require Import Behavior.
Require Import Expr.

Require Import Arith.
Require Import Omega.
Require Import FunctionalExtensionality.

Local Open Scope behavior.
Local Open Scope tla.

Theorem tla_inv :
  ∀ (A : Type) (Init P : Predicate A) (Next : Action A),
    (∀ x, Init x → P x) →
    (∀ x y, P x → Next x y → P y) →
    valid (`Init `/\ [][Next] `=> [] `P).
Proof.
  unfold valid. simpl. unfold shift. simpl.
  intros A Init P Next Hinv1 Hinv2 s (Hini,Hnxt) n.
  induction n; auto.
  specialize (Hnxt n).
  destruct Hnxt as [H | H].
  apply Hinv2 with (s n); auto.
  rewrite ← H; auto.
Qed.

Theorem tla_inv_gen :
  ∀ (A : Type) (Init : Predicate A) (Next : Action A) (F P : Expr A),
    (valid (`Init `∧ [][Next] `∧ F `=> P)) →
    (∀ s, eval P s → Next (s 0) (s 1) → eval P (s @ 1)) →
    valid (`Init `∧ [][Next] `∧ F `=> [] P).
Proof.
  intros A Init Next F P H H0 s.
  intros H1 n.
  destruct H1 as (H1,(H2,H3)).
  generalize dependent s.
  induction n.
  intros s H1 H2 H3.
  apply H. simpl. rewrite shift_0. repeat split; auto.
  intros s H1 H2 H3.
  assert (R : s @ (S n) = (s @ n) @ 1). extensionality x. unfold shift.
  assert (R' : x + S n = x + 1 + n). omega.
  rewrite R'. trivial.
  rewrite R. clear R.
  assert (H4 : Next (s @ n 0) (s @ n 1) ∨ s @ n 0 = s @ n 1). apply H2.
  destruct H4 as [H4|H4].
  apply H0; trivial.
  apply IHn; trivial.
  apply stutter_equiv_shift_1 in H4.
  rewrite contraction in *. simpl in H4.
  assert (H5 : eval P (s @ (S n)) ↔ eval P (s @ n)). apply stutter_equiv_eval;
auto.
  simpl. apply H5. apply IHn; auto.
Qed.
```

C.5 Refinement.v

```
Set Implicit Arguments.

Require Import Behavior.
Require Import Expr.

Local Open Scope behavior.
Local Open Scope tla.

Definition refines (A C : Type)
  (m : C → A)
  (P : Expr A)
  (Q : Expr C)
  := valid (Q `=> map m P).
```

```
Theorem refines_refl :
  ∀ (A : Type) (P : Expr A),
    refines (fun x ⇒ x) P P.
Proof.
  unfold refines, valid. simpl.
  intros.
  rewrite map_id.
  assumption.
Qed.

Theorem refines_trans :
  ∀ (A B C : Type) (m1 : B → A) (m2 : C → B) P Q R,
    refines m1 P Q →
    refines m2 Q R →
    refines (fun x ⇒ m1 (m2 x)) P R.
Proof.
  unfold refines, valid. simpl.
  intros A B C m1 m2 P Q R HQP HRQ s H.
  rewrite ← map_map.
  rewrite map_eval.
  apply HQP.
  rewrite ← map_eval.
  apply HRQ.
  assumption.
Qed.

Theorem refinement_preserves_invariant :
  ∀ (A C : Type)
          (m : C → A)
          (P : Expr A)
          (Q : Expr C)
          (I : Predicate A),
    refines m P Q →
    valid (P ‘=> [] ‘I) →
    valid (Q ‘=> [] map m ‘I).
Proof.
  unfold refines, valid. simpl. unfold shift. simpl.
  intros A C m P Q I Hrefines Hinv s H n.
  specialize (Hinv (fun x ⇒ m (s x))).
  apply Hinv.
  rewrite ← map_eval.
  intuition.
Qed.

Lemma refine_conj :
  ∀ (A C : Type) s (m : C → A) Pa Pc Qa Qc,
    eval (Qc ‘=> map m Qa) s →
    eval (Pc ‘=> map m Pa) s →
    eval (Pc ‘/\ Qc) s →
    eval (map m (Pa ‘/\ Qa)) s.
Proof. firstorder. Qed.

Theorem refine_predicate :
  ∀ (A C : Type) (m : C → A) (Pa : Predicate A) (Pc : Predicate C),
    (∀ x, Pc x → Pa (m x)) →
    refines m ‘Pa ‘Pc.
Proof. firstorder. Qed.

Theorem refine_always :
  ∀ (A C : Type) (m : C → A) P Q,
    refines m P Q →
    refines m ([]P) ([]Q).
Proof. firstorder. Qed.
Theorem refine_always_box :
  ∀ (A C : Type) (m : C → A) (ra : Action A) (rc : Action C),
    (∀ x y, rc x y → ra (m x) (m y)) →
    refines m ([][ra]) ([][rc]).
Proof.
  intros.
  unfold refines, valid. intros s. simpl.
  intros H0 n. specialize H0 with n.
  destruct H0 as [H0|H0]; [left; auto | right; rewrite H0; trivial].
Qed.

Lemma disj_pw_impl :
  ∀ P1 Q1 P2 Q2 : Prop,
    (P1 → P2) →
    (Q1 → Q2) →
    (P1 ∨ Q1) →
    (P2 ∨ Q2).
Proof. firstorder. Qed.
```

C.6 Rules.v

```
Set Implicit Arguments.

Require Import Arith.
Require Import FunctionalExtensionality.
Require Import Omega.
Require Import Setoid.

Require Import Behavior.
Require Import Expr.
Require Import ExprEquiv.

Local Open Scope behavior.
Local Open Scope tla.

Hint Unfold valid.
```

```
Theorem STL1 :
  ∀ (A : Type) (P : Expr A),
    valid P → valid ([] P).
Proof.
  firstorder.
Qed.

Theorem STL2 :
  ∀ (A : Type) (P : Expr A),
    valid ([]P `=> P).
Proof.
  unfold valid. simpl.
  intros A P s H.
  rewrite ← shift_0.
  apply H.
Qed.

Theorem STL3 :
  ∀ (A : Type) (P : Expr A),
    valid ([]([]P) `<=> []P).
Proof.
  unfold valid. simpl.
  intros A P s. split.
  intros H n; specialize (H n 0); rewrite shift_0 in H; assumption.
  intros H n m; specialize (H (m + n)); rewrite contraction; assumption.
Qed.

Theorem STL4 :
  ∀ (A : Type) (P Q : Expr A),
    valid (P `=> Q) →
    valid ([]P `=> []Q).
Proof.
  firstorder.
Qed.

Theorem STL5 :
  ∀ (A : Type) (P Q : Expr A),
    valid ([] (P `/\ Q) `<=> []P `/\ []Q).
Proof.
  firstorder.
Qed.

Theorem STL6 :
  ∀ (A : Type) (P Q : Expr A),
    valid (<>[] (P `/\ Q) `<=> <>[]P `/\ <>[]Q).
Proof.
  split; try (solve [firstorder]).
  intros ((n,HP),(m,HQ)).
  ∃ (n + m). intros o. split.
  specialize (HP (m+o)).
  replace ((s@(n+m))@o) with ((s@n)@(m+o))
    by (unfold shift; extensionality t; intuition).
  assumption.

  specialize (HQ (n+o)).
  replace (s@(n+m)@o) with ((s@m)@(n+o))
    by (unfold shift; extensionality t; intuition).
  assumption.
Qed.

Theorem ImplicationIntroduction : ∀ (A : Type) (P Q : Expr A) s,
                                   (eval P s → eval Q s)
                                   → eval (P ‘=> Q) s.
Proof.
  auto.
Qed.

Theorem ModusPonens : ∀ (A : Type) (P Q : Expr A) s, (eval (P ‘=> Q) s) → eval
P s → eval Q s.
Proof.
  auto.
Qed.

Theorem RefineWF : ∀ (A B : Type) (mapping : A → B) (P : Action A) (Q : Action B),
  (∀ s s’, P s s’ → Q (mapping s) (mapping s’))
  → (∀ s s’, mapping s = mapping s’ → ¬ P s s’ ∨ s = s’)
  → (∀ s, (∃ s’, Q (mapping s) s’) → (∃ s’, P s s’))
  → ∀ s, eval (WF P) s → eval (WF Q) (fun x ⇒ mapping (s x)).
Proof.
  intros A B mapping P Q H diseq H2 s H0.
  assert (H1 : ∀ s, eval (ENABLED P) s → eval (ENABLED Q) (fun x ⇒
mapping (s x))).
    intros s0 H1. destruct H1 as (s1,H1).
    ∃ (mapping s1). apply H; auto.

  assert (H3 : ∀ s, ¬ (∃ s’, P s s’) → ¬ (∃ s’, Q (mapping s) s’)).
  intros st N R. apply N. apply H2. auto.

  assert (wfP : eval (WF P) s). assumption.
  destruct wfP as [wfP|noAct].
  left. intros n. simpl in wfP. specialize wfP with n.
  destruct wfP as (m,(Pm,dif)).
  ∃ m. split. apply H. simpl. apply Pm.
  intros H4. apply diseq in H4. destruct H4; auto.

  right. intros n. simpl in noAct. specialize noAct with n.
  destruct noAct as (n0,noAct). ∃ n0. apply H3; auto.
Qed.

Theorem DeriveAlwaysEventually : ∀ (A : Type) (P Q : Expr A) s,
  (∀ n, eval P (s @ n) → eval (P ‘\/ Q) (s @ (S n))) →
  (∀ n, eval (<> Q) (s @ n) ∨ ¬ eval (<> Q) (s @ n)) →
  eval ([] (P ‘=> ([] P) ‘\/ (<> Q))) s.
Proof.
  intros A P Q s H class n H0.
  assert (T : ∀ m, (∀ k, k ≤ m → eval P ((s @ n) @ k)) ∨ eval (<> Q) (s @ n)).
  intros m. induction m.
  left. intros k H1. inversion H1. rewrite shift_0. auto.
  destruct IHm as [IHm|IHm].
  assert (H1 : eval P (s @ (m + n))). rewrite ← contraction. apply IHm;
auto.
  apply H in H1. destruct H1 as [H1|H1].
  left. intros k H2. inversion H2. rewrite contraction. simpl. auto.
  apply IHm; auto.
  right. ∃ (S m). rewrite contraction. auto.
  right; auto. specialize class with n.
  destruct class. right; auto.
  left. intros m. specialize T with m. destruct T as [T|T].
  apply T; auto.
  exfalso. apply H1. auto.
Qed.

Theorem mapEventuall : ∀ T (P Q : Expr T), valid (P ‘=> Q) →
  valid (<> P ‘=> ≠ Q).
Proof.
  intros T P Q H s eP.
  simpl in *. destruct eP as (n,eP).
  ∃ n. apply H; auto.
Qed.

Theorem deriveLeadsTo' : ∀ T (P Q : Expr T) s,
  (∀ n, eval P (s @ n) → eval (P ‘\/ Q) (s @ (S n)))
  → eval (<> ‘~ P) s → eval (P ‘=> ≠ Q) s.
Proof.
  intros T P Q s H neP.
  assert (H0 : eval P s → ∀ m, (∀ k, k ≤ m → eval P (s @ k)) ∨ eval (<> Q)
s).
  intros H0 m. induction m. left; intros k H1. inversion H1. rewrite
shift_0; auto.
  destruct IHm as [IHm|IHm].
  assert (H1 : eval P s @ m). apply IHm; auto.
  apply H in H1. destruct H1 as [H1|H1].
  left. intros k H2. inversion H2; auto.
  right. ∃ (S m); auto.
  right; auto.
  intros H1. destruct neP as (m,neP).
  assert (H2 : (∀ k : nat, k ≤ m → eval P s @ k) ∨ eval (<> Q) s). auto.
  destruct H2 as [H2|H2] ; auto.
  simpl in *. exfalso. apply neP. apply H2; auto.
Qed.
```

```
Theorem deriveLeadsTo : ∀ T (P Q : Expr T) s,
  (∀ n, eval P (s @ n) → eval (P `\/ Q) (s @ (S n)))
  → eval ([] ≠ `~ P) s → eval (P `~> Q) s.
Proof.
  intros T P Q s H neP n.
  apply deriveLeadsTo'; auto.
  intros m H0.
  rewrite contraction. simpl. apply H.
  rewrite ← contraction. auto.
Qed.
```

C.7 ValidEquiv.v

```
Require Import Expr.
```

```
Definition valid_equiv {A : Type} (P Q : Expr A) : Prop :=
  valid (P `<=> Q).
```

```
Require Import Relations.
Require Import Setoid.
Require Import Morphisms.
Import RelationClasses.
```

```
Lemma valid_equiv_refl :
  ∀ A, reflexive _ (@valid_equiv A).
Proof. firstorder. Qed.
```

```
Lemma valid_equiv_sym :
  ∀ A, symmetric _ (@valid_equiv A).
Proof. firstorder. Qed.
```

```
Lemma valid_equiv_trans :
  ∀ A, transitive _ (@valid_equiv A).
Proof. firstorder. Qed.
```

```
Add Parametric Relation A : (Expr A) (@valid_equiv A)
    reflexivity proved by (valid_equiv_refl A)
    symmetry proved by (valid_equiv_sym A)
    transitivity proved by (valid_equiv_trans A)
      as valid_equiv_rel.
```

```
Add Parametric Morphism A : (@valid A) with
    signature valid_equiv ==> iff as valid_mor.
Proof. firstorder. Qed.
```

```
Add Parametric Morphism A : (@E_Neg A) with
    signature valid_equiv ==> valid_equiv as valid_neg_mor.
Proof. firstorder. Qed.
```

```
Add Parametric Morphism A : (@E_Conj A) with
      signature valid_equiv ==> valid_equiv ==> valid_equiv as valid_conj_mor.
Proof. firstorder. Qed.
Add Parametric Morphism A : (@E_Disj A) with
      signature valid_equiv ==> valid_equiv ==> valid_equiv as valid_disj_mor.
Proof. firstorder. Qed.
Add Parametric Morphism A : (@E_Impl A) with
      signature valid_equiv ==> valid_equiv ==> valid_equiv as valid_impl_mor.
Proof. firstorder. Qed.
Add Parametric Morphism A : (@E_Equiv A) with
      signature valid_equiv ==> valid_equiv ==> valid_equiv as valid_equiv_mor.
Proof. firstorder. Qed.
Add Parametric Morphism A : (@E_Always A) with
      signature valid_equiv ==> valid_equiv as valid_always_mor.
Proof. firstorder. Qed.
Add Parametric Morphism A : (@E_Eventually A) with
      signature valid_equiv ==> valid_equiv as valid_eventually_mor.
Proof. firstorder. Qed.
Module Examples.
  Local Open Scope tla.
  Variable A : Type.
  Implicit Types P Q : Expr A.
  Goal ∀ P Q, valid_equiv P Q → valid_equiv Q P.
    intros. symmetry. assumption.
  Qed.
  Goal ∀ P Q, valid_equiv P Q → valid P → valid Q.
  Proof.
    intros. rewrite ← H. assumption.
  Qed.
  Goal
    ∀ P Q,
      valid_equiv P Q →
      ∀ R, valid ([] P ‘/\ R) → valid ([] Q ‘/\ R).
  Proof.
    intros P Q Heq.
    setoid_rewrite ← Heq.      trivial.
  Qed.
End Examples.
```

C.8 TLA.v

```
Require Export Behavior Expr Invariant Refinement.
```

D Implementation of the AWG in $\mathrm{TLA}^{\mathrm{Coq}}$

D.1 Basic.v

```
Require Import TLA.
Require Import Rules.
Require Import Arith.
Require Import ZArith.
Require Import Omega.
Require Import Setoid.
Require Import FunctionalExtensionality.

Open Scope Z.
Open Scope tla.

Definition ignore {A : Type} (x : A) := True.

Module Type BasicParams.
  Parameter PatternIsh : Type.
  Parameter IsPattern : PatternIsh → Prop.
  Parameter NilPattern : PatternIsh.
  Parameter NilPatternPattern : IsPattern NilPattern.
  Parameter EqNilPatternClassical : ∀ p, IsPattern p → p = NilPattern ∨ p ≠
NilPattern.
End BasicParams.

Inductive MemStatus : Type := Valid | InValid.

Module Basic (Params : BasicParams).
  Import Params.
  Hint Resolve NilPatternPattern.

  Record St := {
      ready : bool;
      status : MemStatus;
      memory : PatternIsh;
      output : PatternIsh}.
  Section Model.
    Variable st st' : St.

    Let ready' := (st').(ready). Let ready := st.(ready).
    Let status' := (st').(status). Let status := st.(status).
    Let memory' := (st').(memory). Let memory := st.(memory).
    Let output' := (st').(output). Let output := st.(output).

    Definition TypeInvariant :=
      IsPattern memory
      ∧ IsPattern output.

    Definition Initialize :=
      ready = false
      ∧ ready' = true
      ∧ match status' with
          | Valid ⇒ memory' = NilPattern
          | _ ⇒ memory = memory' end
      ∧ output = output'.

    Definition Reset :=
      ready' = false
      ∧ IsPattern memory'
      ∧ output' = NilPattern
      ∧ ignore st.

    Definition Flush :=
      ready = true
      ∧ status = Valid
      ∧ memory' = NilPattern
      ∧ output' = NilPattern
      ∧ ready = ready' ∧ status = status'.

    Definition Write p :=
      ready = true
      ∧ status = Valid
      ∧ (p ≠ NilPattern → memory' = p)
      ∧ (p = NilPattern → memory' = memory)
      ∧ ready' = ready ∧ status' = status ∧ output' = output.

    Definition Run :=
      ready = true
      ∧ status = Valid
      ∧ (memory ≠ NilPattern → (output' = memory ∧ memory' = NilPattern))
      ∧ (memory = NilPattern → (output' = output ∧ memory' = memory))
      ∧ ready' = ready ∧ status' = status.

    Definition PowerOn :=
      ready = false
      ∧ output = NilPattern
      ∧ IsPattern memory.

    Definition Next :=
      Initialize
      ∨ Reset
      ∨ Flush
      ∨ (∃ p, IsPattern p ∧ Write p)
      ∨ Run.

  End Model.

  Definition Fairness := WF Initialize.

  Definition Spec := ‘PowerOn ‘∧ [][Next] ‘∧ Fairness.

  Definition ResetEnabled := ENABLED Reset.

  Definition CommandEnabled := ENABLED Run
```

```
                                                    ‘∧ ENABLED Flush
                                                    ‘∧ FORALL (fun p ⇒
                                                              (‘ (fun _ ⇒ IsPat-
tern p))
                                                              ‘=> ENABLED (fun
st st’ ⇒ Write st st’ p)).
```

Definition *CommandsEnabledIffValid* := *CommandEnabled* ‘<=> ‘ (fun *st* ⇒ *st*.(*ready*) = *true* ∧ *st*.(*status*) = *Valid*).

Lemma *Next_TypeInvariant* : ∀ *s s’*, Next *s s’* → *TypeInvariant s* → *TypeInvariant s’*.

```
Proof.
  intros s s’ H H0.
  destruct H0 as (H0,H1).
  destruct H as [(H,(H2,(H3,H4)))
             |[(H,(H2,(H3,_)))
              |[(H,(H2,(H3,(H4,(H5,H6)))))
               |[(p,(H,(H2,(H3,(H4,(H5,(H6,(H7,H8))))))))
                |(H,(H2,(H3,(H4,(H5,H6)))))]]]]; split; auto.
  destruct (status s’).
     rewrite H3; auto. rewrite ← H3; auto.
  rewrite ← H4; auto.
  rewrite H3; auto.
  rewrite H3; auto.
  rewrite H4; auto.
  pose (H9 := EqNilPatternClassical p H).
  destruct H9 as [H9|H9]. apply H5 in H9. rewrite H9; auto.
  apply H4 in H9. rewrite H9; auto.
  rewrite H8; auto.
  pose (H7 := EqNilPatternClassical (memory s) H0).
  destruct H7 as [H7|H7]. apply H4 in H7.
  destruct H7 as (H7,H8). rewrite H8; auto.
  apply H3 in H7. destruct H7 as (H7,H8).
  rewrite H8. auto.
  pose (H7 := EqNilPatternClassical (memory s) H0).
  destruct H7 as [H7|H7]. apply H4 in H7.
  destruct H7 as (H7,H8). rewrite H7; auto.
  apply H3 in H7. destruct H7 as (H7,H8).
  rewrite H7. auto.
Qed.
```

Theorem *Spec_then_TypeInvariant* : *valid* (*Spec* ‘=> [] ‘*TypeInvariant*).

```
Proof.
  apply tla_inv_gen.
  intros s H.
  destruct H as ((H,(H0,H1)),_).
  repeat split; auto. rewrite H0; auto.
```

```
    intros s H H0.
    simpl.
    assert (R: s @ 1%behavior 0%nat = s 1%nat). auto.
    rewrite R.
    apply Next_TypeInvariant with (s 0%nat); auto.
  Qed.

  Theorem Spec_then_ResetEnabled : valid (Spec ‘=> [] ResetEnabled).
  Proof.
    intros s H.
    apply ModusPonens with (‘PowerOn ‘/\ [][Next]).
    apply tla_inv.
    intros x H0. unfold Reset, PowerOn, ignore in *. ∃ x.
    destruct H0 as (H0,(H1,H2)). rewrite H1. repeat split; trivial.
    intros x y H0 H1. destruct H0 as (st’,H0).
    ∃ st’. destruct H0 as (H0,(H2,(H4,H5))). repeat split; trivial.
    destruct H as (H,(H1,H2)).
    split; trivial.
  Qed.

  Lemma Next_CommandsEnabledIffValid : ∀ s, TypeInvariant (s 0%nat) →
                                                  eval CommandsEnabled-
IffValid s →
                                                  Next (s 0%nat) (s 1%nat)
→
                                                  eval CommandsEnabled-
IffValid (s @ 1).
  Proof.
    intros s H H0 H1.
    simpl in *.
    destruct H as (is_pat_mem,is_pat_out).
    destruct H0 as (H0,H2).
    split.
    intros H.
    destruct H as ((st’,(H,(H3,_))),_). split; auto.
    intros H.
    destruct H as (ready_next_true,status_next_valid).
    assert (H3 : TypeInvariant (s 1%nat)). apply Next_TypeInvariant with (s
0%nat); auto; split; auto.
    destruct H3 as (is_pat_next_mem,is_pat_next_out).
    assert (H3 : memory (s @ 1%behavior 0%nat) = NilPattern ∨ memory (s @
1%behavior 0%nat) ≠ NilPattern). apply EqNilPatternClassical; auto.
    repeat split.
    unfold Run. destruct H3 as [H3|H3]. rewrite H3.
    set (nst := {|
      ready := ready (s @ 1%behavior 0%nat);
      status := status (s @ 1%behavior 0%nat);
```

```
        memory := NilPattern;
        output := output (s @ 1%behavior 0%nat)|}).
    ∃ nst; repeat split; unfold nst; auto; intuition.
    set (nst := {|
        ready := ready (s @ 1%behavior 0%nat);
        status := status (s @ 1%behavior 0%nat);
        memory := NilPattern;
        output := memory (s @ 1%behavior 0%nat)|}).
    ∃ nst; repeat split; unfold nst; auto; intuition.
    unfold Flush.
    set (nst := {|
        ready := ready (s @ 1%behavior 0%nat);
        status := status (s @ 1%behavior 0%nat);
        memory := NilPattern;
        output := NilPattern|}).
    ∃ nst; repeat split; unfold nst; auto.
    intros x H.
    unfold Write.
    assert (H4 : x = NilPattern ∨ x ≠ NilPattern). apply EqNilPatternClassical;
auto.
    destruct H4 as [H4|H4].
    set (nst := {|
        ready := ready (s @ 1%behavior 0%nat);
        status := status (s @ 1%behavior 0%nat);
        memory := memory (s @ 1%behavior 0%nat);
        output := output (s @ 1%behavior 0%nat)|}).
    ∃ nst; repeat split; unfold nst; auto; intuition.
    set (nst := {|
        ready := ready (s @ 1%behavior 0%nat);
        status := status (s @ 1%behavior 0%nat);
        memory := x;
        output := output (s @ 1%behavior 0%nat)|}).
    ∃ nst; repeat split; unfold nst; auto; intuition.
  Qed.

  Theorem Spec_then_CommandsEnabledIffValid : valid (Spec ‘=> [] CommandsEn-
abledIffValid).
  Proof.
    assert (G : valid (Spec ‘=> [] (CommandsEnabledIffValid ‘/\ ‘TypeInvariant))).
    apply tla_inv_gen.
    intros s H.
    split. simpl. simpl in H. destruct H as ((H,(H0,H1)),(H2,H3)).
    rewrite H.
    split; intros H4. destruct H4 as ((st’,(H4,_)),_).
    rewrite H4 in H. inversion H.
    destruct H4 as (H4,_). inversion H4.
    assert (H0 : ∀ n, eval ‘ TypeInvariant (s @ n)). apply Spec_then_TypeInvariant;
auto.
    specialize H0 with 0%nat. auto.
    intros s H H0.
    destruct H as (H,H1).
    split.
    apply Next_CommandsEnabledIffValid; auto.
    apply Next_TypeInvariant with (s 0%nat); auto.
    intros s H n.
    apply G in H.
    simpl in H.
    specialize H with n. destruct H as (H,H1).
    auto.
  Qed.
End Basic.
```

D.2 Memory.v

```
Require Import TLA.
Require Import Rules.
Require Import Arith.
Require Import ZArith.
Require Import Omega.
Require Import Setoid.
Require Import FunctionalExtensionality.
Require Import Basic.
Require Import Rules.
Open Scope tla.
Inductive Mode := Record | Play.
Module Type MemParams <: BasicParams.
  Parameter PatternIsh : Type.
  Parameter IsPattern : PatternIsh → Prop.
  Parameter NilPattern : PatternIsh.
  Parameter NilPatternPattern : IsPattern NilPattern.
  Parameter EqNilPatternClassical : ∀ p, IsPattern p → p = NilPattern ∨ p ≠
NilPattern.
  Parameter MaxMemoryLen : nat.
  Parameter MaxMemoryLenGtZ : (0 < MaxMemoryLen)%nat.
  Fixpoint IsListPattern (ls : list PatternIsh) : Prop :=
    match ls with
      | nil ⇒ True
      | (cons h tl) ⇒ IsPattern h ∧ IsListPattern tl
    end.
End MemParams.
```

```
Module ParamsFromMem (Params : MemParams) <: BasicParams.
  Definition PatternIsh := list Params.PatternIsh.
  Definition IsPattern := Params.IsListPattern.
  Definition NilPattern : PatternIsh := nil.
  Theorem NilPatternPattern : IsPattern NilPattern.
  Proof.
    unfold IsPattern, NilPattern.
    simpl. auto.
  Qed.
  Theorem EqNilPatternClassical : ∀ p, IsPattern p → p = NilPattern ∨ p ≠ NilPattern.
  Proof.
    intros p H.
    unfold NilPattern, IsPattern.
    destruct p. left; auto. right; intros F; inversion F.
  Qed.
End ParamsFromMem.

Module Memory (Params : MemParams).
  Import Params.

  Hint Resolve NilPatternPattern.

  Record St := {
      ready : bool;
      status : MemStatus;
      mode : Mode;
      memory : list PatternIsh;
      output : list PatternIsh}.

  Section Model.
    Variable st st' : St.
    Let ready' := (st').(ready). Let ready := st.(ready).
    Let status' := (st').(status). Let status := st.(status).
    Let mode' := (st').(mode). Let mode := st.(mode).
    Let memory' := (st').(memory). Let memory := st.(memory).
    Let output' := (st').(output). Let output := st.(output).

    Definition TypeInvariant := IsListPattern memory ∧ IsListPattern output.

    Definition Initialize :=
      ready = false
      ∧ ready' = true
      ∧ (mode = Play → output = nil)
      ∧ match status' with
            | Valid ⇒ mode' = Record ∧ memory' = nil
            | _ ⇒ mode' = mode ∧ memory' = memory
          end
      ∧ output' = output.

    Definition Reset :=
      ready' = false
```

```
   ∧ memory' = nil
   ∧ output' = nil
   ∧ ignore st.
Definition Flush :=
   ready = true
   ∧ status = Valid
   ∧ mode = Record
   ∧ memory' = nil
   ∧ output' = nil
   ∧ ready' = ready
   ∧ status' = status
   ∧ mode' = mode.
Definition Write p :=
   ready = true
   ∧ status = Valid
   ∧ mode = Record
   ∧ (length memory < MaxMemoryLen)%nat
   ∧ (p ≠ NilPattern → memory' = cons p memory)
   ∧ (p = NilPattern → memory' = memory)
   ∧ ready' = ready
   ∧ status' = status
   ∧ mode' = mode
   ∧ output' = output.
Definition StartPlay :=
   ready = true
   ∧ status = Valid
   ∧ mode = Record
   ∧ output = nil
   ∧ memory ≠ nil
   ∧ mode' = Play
   ∧ ready' = ready ∧ status' = status ∧ memory' = memory ∧ output' = output.
Definition DoPlay :=
   ready = true
   ∧ status = Valid
   ∧ mode = Play
   ∧ ∃ h t, memory = cons h t
   ∧ memory' = t
   ∧ output' = cons h output
   ∧ ready' = ready ∧ status'= status ∧ mode' = mode.
Definition EndPlay :=
   ready = true
   ∧ status = Valid
   ∧ mode = Play
   ∧ memory = nil
```

```
      ∧ mode' = Record
      ∧ output' = output
      ∧ ready' = ready ∧ status' = status ∧ memory' = memory.

    Definition PowerOn :=
      ready = false
      ∧ IsListPattern memory
      ∧ (length memory ≤ MaxMemoryLen)%nat
      ∧ output = nil.

    Definition Next :=
      Initialize
      ∨ Reset
      ∨ Flush
      ∨ (∃ p, IsPattern p ∧ Write p)
      ∨ StartPlay
      ∨ DoPlay
      ∨ EndPlay.
  End Model.

  Definition Fairness :=
     WF Initialize
     `/\ WF DoPlay
     `/\ WF EndPlay.

  Definition Spec := ` PowerOn `/\ [][Next] `/\ Fairness.

  Lemma Next_TypeInvariant : ∀ st st', TypeInvariant st
       → Next st st' → TypeInvariant st'.
  Proof.
    intros st st' tst next.
    unfold TypeInvariant in *.
    destruct tst as (memst,patst).
    destruct st, st'. simpl in *.
    destruct next as [
      (H0,(H1,(H2,(H3,H4))))
      |[(H0,(H1,(H2,H3)))
      |[(H0,(H1,(H2,(H3,(H4,(H5,(H6,H7)))))))
      |[(p,(isPatP,(H0,(H1,(H2,(H3,(H4,(H5,(H6,(H7,(H8,H9)))))))))))
      |[(H0,(H1,(H2,(H3,(H4,(H5,(H6,(H7,(H8,H9)))))))))
      |[(H0,(H1,(H2,(h,(t,(H4,(H5,(H6,(H7,(H8,H9))))))))))
      |(H0,(H1,(H2,(H3,(H4,(H5,(H6,(H7,H8))))))))
    ]]]]]];
    simpl in *.
    destruct status1; destruct H3 as (H3,H5); rewrite H5; rewrite H4;
repeat split; auto.
    rewrite H1, H2; repeat split; auto.
    rewrite H3, H4; repeat split; auto.
    rewrite H9; split; auto.
    assert (H : p = NilPattern ∨ p ≠ NilPattern). apply EqNilPatternClassical.
auto.
    destruct H as [H|H]. apply H5 in H. rewrite H. auto.
    apply H4 in H. rewrite H. split; auto.
    rewrite H8, H9; repeat split; auto.
    rewrite H4 in memst. destruct memst as (iPt,memst).
    rewrite H6, H5; repeat split; auto.
    rewrite H8. rewrite H5. repeat split; auto.
  Qed.

  Theorem Spec_TypeInvariant : valid (Spec ‘=> [] ‘ TypeInvariant).
  Proof.
    apply tla_inv_gen.
    intros s H. destruct H as (H,_).
    unfold PowerOn, TypeInvariant in *; simpl in *.
    destruct H as (_,(H,(_,H0))). rewrite H0. repeat split; auto.
    intros s. apply Next_TypeInvariant.
  Qed.
  Module FROMMEM := ParamsFromMem (Params).
  Module BASIC := Basic.Basic (FROMMEM).

  Definition refinement_mapping (st : St) : BASIC.St :=
    match st with
      | {| ready := ready0;
             status := status0;
             memory := memory0;
             mode := mode0;
             output := output0 |} ⇒
               {| BASIC.ready := ready0;
                  BASIC.status := status0;
                  BASIC.memory := match mode0 with
                                    | Record ⇒ memory0
                                    | Play ⇒ ((List.rev output0) ++ memory0)%list
                                    end;
                  BASIC.output := match mode0 with
                                    | Record ⇒ List.rev output0
                                    | Play ⇒ nil
                                    end |}
    end.

  Lemma Next_refinement : ∀ st st’, TypeInvariant st → Next st st’ →
      (BASIC.Next (refinement_mapping st) (refinement_mapping st’) ∨
      (refinement_mapping st = refinement_mapping st’)).
  Proof.
    intros st st’ typInv next.
    unfold Next, BASIC.Next in *.
    destruct next as [
      (H0,(H1,(H2,(H3,H4))))
      |[(H0,(H1,(H2,H3)))
      |[(H0,(H1,(H2,(H3,(H4,(H5,(H6,H7)))))))
      |[(p,(isPatP,(H0,(H1,(H2,(H3,(H4,(H5,(H6,(H7,(H8,H9)))))))))))
      |[(H0,(H1,(H2,(H3,(H4,(H5,(H6,(H7,(H8,H9)))))))))
      |[(H0,(H1,(H2,(h,(t,(H4,(H5,(H6,(H7,(H8,H9))))))))))
      |(H0,(H1,(H2,(H3,(H4,(H5,(H6,(H7,H8))))))))
    ]]]]].

    left. left.
    unfold BASIC.Initialize.
    destruct st, st'; simpl in *. rewrite ← H0, H1, H4.
    destruct status1; repeat split; auto; destruct H3 as (H3,H5);
    try (rewrite H3); try (rewrite H5); auto.
    destruct mode0; auto; intuition; rewrite H; auto.

    left. right. left.
    unfold BASIC.Reset, ignore.
    destruct st, st'; simpl in *. rewrite H0, H1, H2; simpl.
    destruct mode1; repeat split; auto.

    left. right. right. left.
    unfold Flush, BASIC.Flush in *.
    destruct st, st'; simpl in *.
    rewrite H7.
    rewrite H6, H5, H4, H3, H2, H1. repeat split; auto.

    left. right. right. right. left.
    unfold Write, BASIC.Write in *.
    assert (H10 : p = NilPattern ∨ p ≠ NilPattern). apply EqNilPatternClassical;
auto.
    destruct st, st'; simpl in *.
    rewrite H8. rewrite H0, H1, H2, H6, H7, H9, H1.
    destruct H10 as [H10|H10].
    apply H5 in H10.
    rewrite H10. ∃ FROMMEM.NilPattern; intuition.
    apply H4 in H10. rewrite H10.
    destruct typInv as (ty,_). simpl in ty.
    ∃ ((p :: memory0)%list). repeat split; auto. intros H; inversion H.

    right.
    destruct st, st'. simpl in *.
    rewrite H9, H8, H7, H6, H5, H3, H2, H1, H0. simpl in *.
    auto.

    right.
    destruct st, st'. simpl in *.
    rewrite H9, H8, H7, H6, H5, H4, H2, H1, H0.
    simpl.
    rewrite ← List.app_assoc.
    simpl. auto.
    left. right. right. right. right.
    unfold BASIC.Run.
    destruct st, st'; simpl in *.
    rewrite H8, H7, H6, H5, H4, H3, H2, H1, H0.
    repeat split; auto.
    rewrite List.app_nil_r. auto.
    rewrite List.app_nil_r in H. auto.
  Qed.
End Memory.
```

D.3 Command.v

```
Require Import TLA.
Require Import Rules.
Require Import Arith.
Require Import ZArith.
Require Import Omega.
Require Import Setoid.
Require Import FunctionalExtensionality.
Require Import Basic.
Require Import Rules.

Open Scope Z.
Open Scope tla.

Definition ignore {A : Type} (x : A) := True.

Module Command (Params : BasicParams).
  Import Params.

  Hint Resolve NilPatternPattern.

  Inductive Control : Type := Ready | Busy | Done.
  Inductive Command : Type :=
  | Flush : Command
  | Write : PatternIsh → Command
  | Run : Command.
  Definition IsCommand (c : Command) : Prop := match c with
                                                 | Write p ⇒ IsPattern p
                                                 | _ ⇒ True
                                               end.

  Record St := {
      ready : bool;
      status : MemStatus;
      ctrl : Control;
      cmd : Command;
      memory : PatternIsh;
      output : PatternIsh}.
```

```
Section Model.
  Variable st st' : St.
  Let ready' := (st').(ready). Let ready := st.(ready).
  Let status' := (st').(status). Let status := st.(status).
  Let ctrl' := (st').(ctrl). Let ctrl := st.(ctrl).
  Let cmd' := (st').(cmd). Let cmd := st.(cmd).
  Let memory' := (st').(memory). Let memory := st.(memory).
  Let output' := (st').(output). Let output := st.(output).
  Definition TypeInvariant :=
    IsCommand cmd
    ∧ IsPattern memory
    ∧ IsPattern output.
  Definition Initialize :=
    ready = false
    ∧ ready' = true
    ∧ ctrl' = Ready
    ∧ match status' with
        | Valid ⇒ memory' = NilPattern
        | _ ⇒ memory = memory' end
    ∧ output = output'
    ∧ cmd = cmd'.
  Definition Reset :=
    ready' = false
    ∧ IsPattern memory'
    ∧ IsCommand cmd'
    ∧ output' = NilPattern
    ∧ ignore st.        Definition Req c :=
    ready = true
    ∧ status = Valid
    ∧ ctrl = Ready
    ∧ cmd' = c
    ∧ ctrl' = Busy
    ∧ ready = ready' ∧ status = status' ∧ memory = memory' ∧ output = output'.
  Definition DoFlush :=
    cmd = Flush
    ∧ memory' = NilPattern
    ∧ output' = NilPattern.
  Definition DoWrite :=
    (∃ p, cmd = Write p
                    ∧ (p ≠ NilPattern → memory' = p)
                    ∧ (p = NilPattern → memory' = memory))
    ∧ output = output'.
  Definition DoRun :=
    cmd = Run
    ∧ (memory ≠ NilPattern → (output' = memory ∧ memory' = NilPattern))
    ∧ (memory = NilPattern → (output' = output ∧ memory' = memory)).

  Definition Do :=
    ready = true
    ∧ status = Valid
    ∧ ctrl = Busy
    ∧ (DoFlush ∨ DoWrite ∨ DoRun)
    ∧ ctrl' = Done
    ∧ (ready' = ready ∧ status' = status ∧ cmd' = cmd).

  Definition Resp :=
    ready = true
    ∧ status = Valid
    ∧ ctrl = Done
    ∧ ctrl' = Ready
    ∧ (ready' = ready
        ∧ status' = status
        ∧ cmd' = cmd
        ∧ memory' = memory
        ∧ output' = output).

  Definition PowerOn :=
    ready = false
    ∧ IsCommand cmd
    ∧ IsPattern memory
    ∧ output = NilPattern.

  Definition Next :=
    Initialize
    ∨ Reset
    ∨ (∃ c, IsCommand c ∧ Req c)
    ∨ Do
    ∨ Resp.

End Model.

Definition Fairness :=
  WF Initialize
  `/\ WF Do
  `/\ WF Resp.

Definition Spec := `PowerOn `/\ [][Next] `/\ Fairness.

Lemma Next_TypeInvariant : ∀ st st', TypeInvariant st → Next st st' → TypeInvariant st'.
Proof.
  intros st st' H H0.
  destruct st.
  destruct st'.
  unfold TypeInvariant in H. simpl in H.
```

```
    destruct H as (IsCmd_cmd0,(IsPat_memy0,IsPat_out0)).
    destruct H0 as [(_,(_,(_,(H0,(out_unchanged,cmd_unchanged)))))
                  |[(_,(IsPat_mem1,(IsPat_cmd1,(out1_nilPat,_))))
                   |[(c,(isCommand_c,(_,(_,(_,(cmd1_c,(_,(_,(_,(mem_unchanged,out_unchanged))))))))))
                    |[(_,(_,(_,([(_,(mem1_nilPat,out1_nilPat))
                                   |[((p,(cmd0_writeP,(p_ifnotnilPat,p_ifNilPat))),out_unchanged)
                                    |(_,(mem0_ifnotnilPat,mem0_ifNilPat))]],(_,(_,(_,cmd_unchanged)))))))
                     |(_,(_,(_,(_,(_,(_,(cmd_unchanged,(mem_unchanged,out_unchanged))))))))]]]];
simpl in *; unfold TypeInvariant; simpl;

    try (rewrite cmd_unchanged in *);
    try (rewrite out_unchanged in *);
    try (rewrite mem_unchanged in *);
    try (rewrite out1_nilPat);
    try (rewrite mem1_nilPat);
    try (rewrite cmd1_c);
    try (rewrite cmd0_writeP in *); repeat split; auto.

    destruct status1; rewrite H0 in *; auto.

    unfold IsCommand in *.
    assert (H : p = NilPattern ∨ p ≠ NilPattern). apply EqNilPatternClassical;
auto.
    destruct H as [H|H].
    apply p_ifNilPat in H; rewrite H; auto.
    apply p_ifnotnilPat in H; rewrite H; auto.

    assert (H : memory0 = NilPattern ∨ memory0 ≠ NilPattern). apply EqNilPat-
ternClassical; auto.
    destruct H as [H|H].
    apply mem0_ifNilPat in H. destruct H as (H,H0). rewrite H0; auto.
    apply mem0_ifnotnilPat in H. destruct H as (H,H0). rewrite H0; auto.

    assert (H : memory0 = NilPattern ∨ memory0 ≠ NilPattern). apply EqNilPat-
ternClassical; auto.
    destruct H as [H|H].
    apply mem0_ifNilPat in H. destruct H as (H,H0). rewrite H; auto.
    apply mem0_ifnotnilPat in H. destruct H as (H,H0). rewrite H; auto.
  Qed.

  Theorem Spec_TypeInvariant : valid (Spec ‘=> [] ‘ TypeInvariant).
  Proof.
    apply tla_inv_gen.
    intros s H.
    destruct H as ((H,(H0,(H1,H2))),_). simpl in *.
    repeat split; auto; rewrite H2; auto.
    intros s H H0. apply Next_TypeInvariant with (s 0%nat); auto.
  Qed.

  Module BASIC := Basic.Basic Params.
```

```
Definition refinement_mapping : St → BASIC.St :=
  fun x ⇒ match x with
              | {| ready := ready0;
                   status := status0;
                   output := output0;
                   memory := memory0 |} ⇒
               {|
                 BASIC.ready := ready0;
                 BASIC.status := status0;
                 BASIC.memory := memory0;
                 BASIC.output := output0 |}
           end.

Lemma refines_Next : ∀ st st',
                       TypeInvariant st
                       → Next st st'
                       → (BASIC.Next
                            (refinement_mapping st)
                            (refinement_mapping st') ∨
                          refinement_mapping st = refinement_mapping st').
Proof.
  intros st st' typeInvSt H.
  destruct typeInvSt as (isCommandSt,(isPatternMemSt,isPatternOutSt)).
  unfold Next in *.
  unfold BASIC.Next in *.
  Ltac break_up := unfold BASIC.Initialize, BASIC.Reset, BASIC.Flush, BA-
SIC.Write, BASIC.Run in *;
    unfold Initialize, Reset, Req, Do, Resp in *.
  destruct H as [H|[H|[(c,(isComc,H))|[H|H]]]].
  left. left. break_up; destruct st, st'; simpl in *.
  destruct H as (H,(H1,(H2,(H3,(H4,H5))))).
  destruct status1; repeat split; auto.
  left. right. left. break_up; destruct st, st'; simpl in *.
  destruct H as (H,(H1,(H2,(H3,H4)))); repeat split; auto.
  right. break_up; destruct st, st'; simpl in *.
  destruct H as (_,(_,(_,(_,(_,(H4,(H5,(H6,H7)))))))).
  rewrite H4, H5, H6, H7; auto.
  destruct st. simpl.
  destruct H as (H,(H0,(H1,([H2|[H2|H2]],(H3,(H4,(H5,H6))))))); destruct
cmd0 as [|p|];
  unfold DoFlush,DoWrite,DoRun in *; simpl in *.
  left. right. right. left.
  break_up; destruct st'; simpl in *.
  destruct H2 as (_,(H2,H7)).
  repeat split; auto.
  destruct H2 as (H2,_); inversion H2.
```

```
    destruct H2 as (H2,_); inversion H2.
    destruct H2 as ((p0,(H2,_)),_); inversion H2.
    destruct H2 as ((p0,(H2,(H7,H8))),H9).
    left. right. right. right. left. ∃ p.
    assert (H10 : p = p0). inversion H2; auto.
    rewrite H10 in *.
    repeat split; auto; destruct st'; simpl in *; auto.
    destruct H2 as ((p0,(H2,_)),_); inversion H2.
    destruct H2 as (H2,_); inversion H2.
    destruct H2 as (H2,_); inversion H2.
    destruct H2 as (_,(H2,H7)).
    left. do 4 right.
    destruct st'; repeat split; simpl in *; auto; intuition.
    right.
    destruct H as (H,(H1,(H2,(H3,(H4,(H5,(H6,(H7,H8)))))))).
    destruct st, st'; simpl in *. rewrite H4, H5, H7, H8. auto.
  Qed.

  Lemma refines_init : ∀ st, PowerOn st → BASIC.PowerOn (refinement_mapping st).
  Proof.
    intros st H.
    unfold PowerOn, BASIC.PowerOn in *.
    destruct H as (H,(H0,(H1,H2))).
    destruct st; simpl in *.
    repeat split; auto.
  Qed.

  Theorem refines : valid (Spec ‘=> (map refinement_mapping BASIC.Spec)).
  Proof.
    intros s H.
    assert (tyInv : eval ([] ‘TypeInvariant) s). apply Spec_TypeInvariant in H;
auto.
    destruct H as (init,(next,fairness)).
    rewrite map_eval.
    unfold BASIC.Spec.
    split. rewrite ← map_eval. apply refines_init; auto.
    split. rewrite ← map_eval. simpl.
    intros n. simpl in next. specialize next with n.
    destruct next as [H|H].
    apply refines_Next; auto; apply tyInv.
    right; rewrite H; trivial.
    assert (H : ∀ s s',Initialize s s'
        → BASIC.Initialize (refinement_mapping s) (refinement_mapping s')).
    intros s0 s1 H. destruct s0. destruct s1.
    unfold Initialize in H. unfold BASIC.Initialize. simpl in *.
    destruct H as (H,(H1,(H2,(H3,(H4,H5))))).
    destruct status1; repeat split; auto.
    apply RefineWF with Initialize.
    assumption.
    intros s0 s1 H0. left. intros H2. destruct H2 as (H2,(H3,H4)).
    destruct s0. destruct s1. simpl in *. rewrite H2 in *.
    rewrite H3 in *. inversion H0.
    intros s0 H1.
    destruct H1 as (s',(H1,(H3,(H4,H5)))).
    ∃ ({|ready := true;
              status := BASIC.status s';
              ctrl := Ready;
              cmd := cmd s0;
              memory := BASIC.memory s';
              output := output s0|}).
    destruct s0, s'. simpl in *.
    repeat split; auto.
    destruct fairness as (WF_init,_); auto.
  Qed.

  Definition ResetEnabled := ENABLED Reset.

  Lemma Next_ResetEnabled : ∀ st st',
                            (∃ s, Reset st s)
                            → Next st st'
                            → ∃ s, Reset st' s.
  Proof.
    intros st st' H next.
    destruct st, st'.
    unfold Reset.
    destruct H as (s,H). ∃ s. unfold Reset in H.
    repeat split; intuition; auto.
  Qed.

  Theorem Spec_ResetEnabled : valid (Spec '=> [] ResetEnabled).
  Proof.
    apply tla_inv_gen.
    intros s H. destruct H as ((H,(H1,(H2,H3))),_).
    simpl.
    ∃ ({|ready := false;
              status := (s 0%nat).(status);
              ctrl := (s 0%nat).(ctrl);
              cmd := (s 0%nat).(cmd);
              memory := (s 0%nat).(memory);
              output := NilPattern|}).
    unfold Reset; simpl; repeat split; auto.
    intros s. apply Next_ResetEnabled.
  Qed.

  Definition IsCommand' {T} c := ' fun (x : T) ⇒ IsCommand c.
```

```
  Definition CommandsEnabled := E_ForallRigid (fun c ⇒ IsCommand' c
                                    '=> ENABLED (fun a b ⇒ Req a b c)).

  Definition CommandsEnabledIffValidAndReady := CommandsEnabled '<=>
                                                   ' fun st ⇒ st.(ready) = true
                                                         ∧ st.(status) = Valid
                                                         ∧ st.(ctrl) = Ready.

  Lemma Next_CommandsEnabledIffValidAndReady : ∀ st st',
      ((∀ c, IsCommand c → ∃ s, Req st s c) ↔ (st.(ready) = true
                                                         ∧ st.(status) = Valid
                                                         ∧ st.(ctrl) = Ready))
      → Next st st'
      → (∀ c, IsCommand c → ∃ s, Req st' s c) ↔ ((st').(ready) = true
                                                         ∧ (st').(status) =
Valid
                                                         ∧ (st').(ctrl) = Ready).
  Proof.
    intros st st' H next.
    split.

    intros H0.
    assert (H1 : ∃ s : St, Req st' s Flush). apply H0; simpl; auto.
    destruct H1 as (s,H1). unfold Req in H1. repeat split; intuition;
auto.

    intros H0 c isCmdC. destruct H0 as (H0,(H1,H2)).
    ∃ ({| ready := ready st';
              status := status st';
              ctrl := Busy;
              cmd := c;
              memory := memory st';
              output := output st'|}).
    repeat split; simpl; auto.
  Qed.

  Theorem Spec_CommandsEnabledIffValidAndReady : valid (Spec '=> [] Command-
sEnabledIffValidAndReady).
  Proof.
    apply tla_inv_gen.
    intros s H. destruct H as (H,_).
    split. intros H1. simpl in H1.
    assert (H2 : ∃ st' : St, Req (s 0%nat) st' Flush). apply H1; auto; unfold
IsCommand; auto.
    destruct H2 as (st', H2). unfold Req in H2. repeat split; intuition;
auto.
    intros H1 c H2. simpl in H2.
    destruct H1 as (H1,(H3,H4)).
    unfold Req.
    ∃ ({| ready := ready (s O);
                 status := status (s O);
                 ctrl := Busy;
                 cmd := c;
                 memory := memory (s O);
                 output := output (s O)|}).
    repeat split; simpl; auto.
    intros s. apply Next_CommandsEnabledIffValidAndReady.
  Qed.

  Definition CommandsComplete := ‘ (fun st ⇒ st.(ctrl) = Busy) ‘~> ‘ (fun st ⇒
st.(ctrl) = Ready).

  Lemma Spec_NotBad : valid (Spec ‘=> [] ‘ fun st ⇒ ¬ (st.(ready) = true
                                                       ∧ st.(status) = In-
Valid
                                                       ∧ (st.(ctrl) = Busy
∨ st.(ctrl) = Done))).
  Proof.
    apply tla_inv_gen.
    intros s H F. destruct H as (H,_).
    destruct H as (H,_). destruct F as (F,_). rewrite F in H. inversion
H.
    intros s H next F.
    destruct F as (rt,(si,cb)). simpl in H.
    destruct next as [(_,(_,(F,_)))
                      |[(F,_)
                      |[(c,(isC,(_,(F0,(_,(_,(_,(_,(F1,_)))))))))
                      |[(_,(F0,(_,(_,(_,(_,(F1,_)))))))
                      |(_,(F0,(_,(_,(_,(F1,_)))))))]]]]; unfold shift in *; simpl
in *.
    destruct cb as [cb|cb].
    assert (W : Busy = Ready). rewrite ← cb; rewrite ← F; auto. inversion
W.
    assert (W : Done = Ready). rewrite ← cb; rewrite ← F; auto. inversion
W.
    assert (W : true = false). rewrite ← rt; rewrite ← F; auto. inversion
W.
    assert (W : InValid = Valid). rewrite ← si; rewrite ← F0; rewrite F1;
auto. inversion W.
    assert (W : InValid = Valid). rewrite ← si; rewrite ← F0; rewrite ←
F1; auto. inversion W.
    assert (W : InValid = Valid). rewrite ← si; rewrite ← F0; rewrite ←
F1; auto. inversion W.
  Qed.

  Definition IsReady st := st.(ctrl) = Ready.
  Definition IsBusy st := st.(ctrl) = Busy.
  Definition IsDone st := st.(ctrl) = Done.
  Definition IsValid st := st.(status) = Valid.
  Definition IsInvalid st := st.(status) = InValid.
  Definition ReadyTrue st := st.(ready) = true.
  Definition ReadyFalse st := st.(ready) = false.

  Lemma Spec_ReadyFalseLeadstoReady : valid (Spec '=> ((' ReadyFalse) '~> ' Is-
Ready)).
  Proof.
    intros s spec.
    assert (H : eval (([]<> ' IsReady) '\/ ([]<> 'ReadyTrue)) s).
    destruct spec as (_,(_,([wfI|wfI],_))); simpl in *; unfold Initialize in *;
    unfold IsReady; unfold ReadyFalse.
    left. intros n. specialize wfI with n. destruct wfI as (m,((_,(_,(wfI,_))),_)).
    ∃ (S m). rewrite ← wfI; auto.
    right. intros n. specialize wfI with n.
    destruct wfI as (m,wfI).
    assert (F : ready ((s @ n) @ m%behavior 0%nat) = true ∨ ready ((s @ n) @
m%behavior 0%nat) = false).
    destruct (ready ((s @ n) @ m%behavior 0%nat)); auto.
    destruct F as [F|F]. ∃ m; auto.
    exfalso. apply wfI.
    ∃ ({| ready := true;
                  ctrl := Ready;
                  status := Valid;
                  memory := NilPattern;
                  output := output ((s @ n) @ m%behavior 0%nat);
                  cmd := cmd ((s @ n) @ m%behavior 0%nat)|}).
    repeat split; simpl; auto.
    destruct H as [H|H].
    simpl in *; intros n; specialize H with n; auto.
    apply deriveLeadsTo.
    intros n H0.
    destruct spec as (_,(spec,_)).
    simpl in *. unfold ReadyFalse in *. unfold IsReady in *. specialize
spec with n.
    destruct spec as [
    [next
    |[next
    |[(c,(_,(H1,_)))
    |[(H1,_)
    |(H1,_)]]]]
   |next].
    firstorder.
    firstorder.
    rewrite H0 in H1. inversion H1.
    rewrite H0 in H1. inversion H1.
    rewrite H0 in H1. inversion H1.
    left. rewrite next in H0. rewrite ← H0. auto.
    clear spec.
    simpl in *. intros n. specialize H with n.
    destruct H as (m,H). ∃ m. unfold ReadyTrue in *. unfold ReadyFalse
in *.
    rewrite H; intros F; inversion F.
  Qed.

  Lemma Spec_ValidAndDoneLeadsToReady : valid (Spec ‘=>
          ((‘ ReadyTrue ‘∧ ‘ IsValid ‘∧ ‘ IsDone) ‘~> ‘ IsReady)).
  Proof.
    intros s spec.
    assert (T : eval (‘ ReadyTrue ‘∧ ‘ IsValid ‘∧ ‘ IsDone ‘~> (‘ IsReady ‘∨ ‘
ReadyFalse)) s
                  → eval (‘ ReadyTrue ‘∧ ‘ IsValid ‘∧ ‘ IsDone ‘~> ‘ IsReady) s).
    intros H n H0.
    apply H in H0.
    destruct H0 as (m,[H0|H0]).
    ∃ m; auto.
    apply Spec_ReadyFalseLeadstoReady in spec.
    apply spec in H0. destruct H0 as (m0,H0).
    ∃ ((m0 + m)%nat). unfold IsReady in *. simpl in *.
    rewrite ← H0. do 2 rewrite ← contraction; auto.
    apply T. clear T.
    assert (H : eval (([]<> ‘ IsReady) ‘∨ ([]<> ‘~ (‘ ReadyTrue ‘∧ ‘ IsValid ‘∧ ‘
IsDone))) s).
    destruct spec as (_,(_,(_,(_,[fair|fair])))));
    unfold IsReady, ReadyTrue, IsValid, IsDone, Resp in *.
    left. intros n. simpl in *. specialize fair with n.
    destruct fair as (m,((_,(_,(_,(H,_)))),_)).
    ∃ (S m). rewrite ← H; auto.
    right. simpl in *. intros n. specialize fair with n.
    destruct fair as (m,fair). ∃ m.
    intros H. destruct H as (H,(H0,H1)).
    destruct (ready ((s @ n) @ m%behavior 0%nat)).
    destruct (status ((s @ n) @ m%behavior 0%nat)).
    destruct (ctrl ((s @ n) @ m%behavior 0%nat)).
    inversion H1. inversion H1.
    apply fair.
    ∃ ({| ctrl := Ready;
                ready := true;
                status := Valid;
                cmd := cmd ((s @ n) @ m%behavior 0%nat);
                memory := memory ((s @ n) @ m%behavior 0%nat);
```

```
                    output := output ((s @ n) @ m%behavior 0%nat) |}). repeat split;
auto.
    inversion H0.
    inversion H.
    destruct H as [H|H].
    clear spec. intros n _. simpl in *. specialize H with n. destruct
H as (m,H).
    ∃ m. left; auto.
    apply deriveLeadsTo; auto.
    intros n H0.
    destruct spec as (_,(next,_)).
    simpl in next. specialize next with n. clear H.
    unfold ReadyTrue, IsValid, IsDone, IsReady, ReadyFalse in *.
    simpl in *.
    destruct H0 as (H0,(H1,H2)).
    destruct next as [
    [(H3,_)
    |[next
    |[(c,(_,(_,(_,(H3,_)))))
    |[(_,(_,(H3,_)))
    |(_,(_,(_,(H3,_))))]]]]
   |next].
    assert (W : true = false). rewrite ← H3. rewrite H0. auto. inversion
W.
    right. right. firstorder.
    assert (W : Ready = Done). rewrite ← H3. rewrite ← H2. auto. inversion
W.
    assert (W : Busy = Done). rewrite ← H3. rewrite ← H2. auto. inversion
W.
    right. left. rewrite ← H3. auto.
    rewrite next in *. rewrite ← H0. rewrite ← H1. rewrite ← H2.
    left; repeat split; auto.
  Qed.

  Lemma Spec_ValidAndBusyLeadsToReady : valid (Spec ‘=>
          ((‘ ReadyTrue ‘/\ ‘ IsValid ‘/\ ‘ IsBusy) ‘~> ‘ IsReady)).
  Proof.
     intros s spec.
     assert (T : eval (‘ ReadyTrue ‘/\ ‘ IsValid ‘/\ ‘ IsBusy
                    ‘~> ((‘ ReadyTrue ‘/\ ‘ IsValid ‘/\ ‘ IsDone) ‘\/ (‘ IsReady ‘\/ ‘
ReadyFalse))) s
                    → eval (‘ ReadyTrue ‘/\ ‘ IsValid ‘/\ ‘ IsBusy ‘~> ‘ IsReady) s).
     intros H n H0. apply H in H0.
     destruct H0 as (m,[H0|[H0|H0]]).
     apply Spec_ValidAndDoneLeadsToReady in spec.
     apply spec in H0. unfold IsReady in *.
```

```
      destruct H0 as (n0,H0). ∃ ((n0 + m)%nat).
      simpl in *. rewrite ← H0. do 2 rewrite ← contraction. auto.
      ∃ m. auto.
      apply Spec_ReadyFalseLeadstoReady in spec. apply spec in H0.
      destruct H0 as (m0,H0). ∃ ((m0 + m)%nat).
      unfold IsReady in *.
      simpl in *. rewrite ← H0. do 2 rewrite ← contraction. auto.
      apply T. clear T.
      assert (H: eval (([]<> (‘ IsReady ‘\/ ‘ ReadyFalse))
                    ‘\/ []<> ‘~ (‘ ReadyTrue ‘/\ ‘ IsValid ‘/\ ‘ IsBusy)) s).
      assert (H0 : eval (‘ ReadyTrue ‘/\ ‘ IsValid ‘/\ ‘ IsDone ‘~> ‘IsReady) s).
      apply Spec_ValidAndDoneLeadsToReady; auto.
      assert (TY : eval ([] ‘ TypeInvariant) s). apply Spec_TypeInvariant in
spec. auto.
      destruct spec as (_,(_,(_,([fair|fair],_)))).
      unfold Do in fair.
      simpl in fair. left. intros n. specialize fair with n.
      destruct fair as (m,fair).
      assert (H1 : eval (‘ ReadyTrue ‘/\ ‘IsValid ‘/\ ‘IsDone) ((s @ n) @ (S m)%behavior)).
      unfold ReadyTrue, IsValid, IsDone. simpl.
      destruct fair as ((H1,(H2,(_,(_,(H3,(H4,(H5,_))))))),_).
      rewrite ← H1. rewrite ← H2. rewrite ← H3. rewrite ← H4. rewrite
← H5.
      repeat split; auto.
      apply H0 in H1. destruct H1 as (m0,H1). ∃ ((m0 + S m)%nat).
      left. unfold IsReady in *. rewrite ← H1.
      unfold eval. do 2 rewrite ← contraction. simpl. auto.
      right. intros n. simpl in fair.
      specialize fair with n. destruct fair as (m,fair).
      clear H0. ∃ m. unfold ReadyTrue, IsValid, IsBusy, Do in *. simpl in *.
      intros F. destruct F as (H,(H0,H1)).
      destruct (ready ((s @ n) @ m%behavior 0%nat)).
      destruct (status ((s @ n) @ m%behavior 0%nat)).
      destruct (ctrl ((s @ n) @ m%behavior 0%nat)).
      inversion H1.
      unfold DoFlush, DoWrite, DoRun in *.
      apply fair.
      assert (IsC : IsCommand (cmd ((s @ n) @ m%behavior 0%nat))).
      specialize TY with ((m + n)%nat). rewrite contraction.
      destruct TY as (TY0,(TY1,TY2)). auto.
      destruct (cmd ((s @ n) @ m%behavior 0%nat)).
      ∃ ({| ctrl := Done;
                ready := true;
                status := Valid;
                cmd := Flush;
```

```
                  output := NilPattern;
                  memory := NilPattern|}).
        repeat split; simpl; auto.
        simpl in IsC.
        apply EqNilPatternClassical in IsC. destruct IsC as [IsC|IsC].
        ∃ ({| ctrl := Done;
                  ready := true;
                  status := Valid;
                  cmd := Write p;
                  output := (output ((s @ n) @ m%behavior 0%nat));
                  memory := (memory ((s @ n) @ m%behavior 0%nat))|}).
        repeat split; simpl; auto. right. left. split; auto. ∃ p. repeat
split; auto.
        intros F. exfalso. apply F; auto.
        ∃ ({| ctrl := Done;
                  ready := true;
                  status := Valid;
                  cmd := Write p;
                  output := (output ((s @ n) @ m%behavior 0%nat));
                  memory := p|}).
        repeat split; simpl; auto. right. left. split; auto. ∃ p. repeat
split; auto.
        intros F. exfalso. apply IsC; auto.
        assert (Q : IsPattern (memory ((s @ n) @ m%behavior 0%nat))).
        specialize TY with ((m + n) % nat). unfold TypeInvariant in TY.
        rewrite contraction. destruct TY as (_,(TY,_)). auto.
        apply EqNilPatternClassical in Q.
        destruct Q as [Q|Q].
        ∃ ({| ctrl := Done;
                  ready := true;
                  status := Valid;
                  cmd := Run;
                  output := output ((s @ n) @ m%behavior 0%nat);
                  memory := memory ((s @ n) @ m%behavior 0%nat)|}).
        repeat split; simpl; auto. right. right. repeat split; auto.
        exfalso; apply H2; auto.
        ∃ ({| ctrl := Done;
                  ready := true;
                  status := Valid;
                  cmd := Run;
                  output := memory ((s @ n) @ m%behavior 0%nat);
                  memory := NilPattern|}).
        repeat split; simpl; auto. right. right. repeat split; auto.
        exfalso; apply Q; auto.
        inversion H1.
```

```
    inversion H0.
    inversion H.
    destruct H as [H|H].
    intros n _. simpl in *. specialize H with n. destruct H as (m,H).
    ∃ m. right. auto.
    apply deriveLeadsTo; auto. clear H.
    intros n H. destruct spec as (_,(next,_)).
    simpl in next. specialize next with n.
    destruct H as (H,(H0,H1)).
    unfold ReadyTrue, IsValid, IsBusy, IsDone, IsReady, ReadyFalse in *.
    simpl in *.
    unfold Next in *.
    destruct next as [
    [(H2,_)
    |[(H2,_)
    |[(c,(_,(_,(_,(H2,_)))))
    |[(H2,(H3,(_,(_,(H4,(H5,(H6,_)))))))
    |(_,(_,(H2,_)))]]]]
  |next].
    assert (W : true = false). rewrite ← H2. rewrite ← H. auto. inversion
W.
    right. right. right. rewrite ← H2. auto.
    assert (W : Ready = Busy). rewrite ← H2. rewrite ← H1. auto. inversion
W.
    right. left. rewrite ← H2. rewrite ← H3. rewrite ← H4.
     rewrite ← H5. rewrite ← H6. repeat split; auto.
    assert (W : Done = Busy). rewrite ← H2. rewrite ← H1. auto. inversion
W.
    rewrite next in *. left. rewrite ← H. rewrite ← H0. rewrite ←
H1.
    repeat split; auto.
Qed.

Theorem Spec_CommandsComplete : valid (Spec '=> CommandsComplete).
Proof.
   intros s spec.
   pose (H1 := Spec_NotBad s spec).
   intros n H0.
   assert (H2 : eval ( (' ReadyTrue '/\ ' IsValid '/\ ' IsBusy)
                    '\/ (' ReadyTrue '/\ ' IsValid '/\ ' IsDone)
                    '\/ ' ReadyFalse '\/ ' IsReady) s @ n).
   unfold IsValid, IsBusy, IsReady, IsDone, ReadyTrue, ReadyFalse in *.
   simpl. simpl in H1. specialize H1 with n.
   destruct (ctrl (s @ n%behavior 0%nat)).
   right; right. right. auto.

   destruct (ready (s @ n%behavior 0%nat)).
```

```
    destruct (status (s @ n%behavior 0%nat)).
    left. repeat split; auto.
    exfalso. apply H1. repeat split; auto.
    right. right. left. auto.

    destruct (ready (s @ n%behavior 0%nat)).
    destruct (status (s @ n%behavior 0%nat)).
    right. left. repeat split; auto.
    exfalso. apply H1. repeat split; auto.
    right. right. left. auto.

    destruct H2 as [H2|[H2|[H2|H2]]].
    pose (H3 := Spec_ValidAndBusyLeadsToReady s spec).
    apply H3; auto.
    pose (H3 := Spec_ValidAndDoneLeadsToReady s spec).
    apply H3; auto.
    pose (H3 := Spec_ReadyFalseLeadstoReady s spec).
    apply H3; auto.
    unfold IsReady in H2. simpl in *.
    ∃ O. rewrite shift_0. auto.
  Qed.

  Definition InitializeAfterReset := ‘ (fun st ⇒ st.(ready) = false)
                                                  ‘~>
                                                  ‘ fun st ⇒ st.(ready) = true.

  Theorem Spec_InitializeAfterReset : valid (Spec ‘=> InitializeAfterReset).
  Proof.
    intros s H.
    destruct H as (init,(next,([wfI|wfI],fairness))).
    intros n H. simpl in wfI. specialize wfI with n.
    destruct wfI as (m,(wfI,H0)).
    ∃ (S m). simpl. unfold Initialize in wfI.
    destruct wfI as (_,(wfI,_)). auto.
    simpl in wfI.
    intros n H. specialize wfI with n.
    destruct wfI as (m,imp).
    ∃ m. simpl.
    assert (B : ready ((s @ n) @ m%behavior 0%nat) = true ∨ ready ((s @ n) @
m%behavior 0%nat) = false).
    destruct (ready ((s @ n) @ m%behavior 0%nat)); intuition.
    destruct B as [B|B]; auto.
    exfalso. apply imp. unfold Initialize.
    ∃ ({| ready := true;
              ctrl := Ready;
              status := Valid;
              memory := NilPattern;
              output := output ((s @ n) @ m%behavior 0%nat);
```

```
                cmd := cmd ((s @ n) @ m%behavior 0%nat) |}).
    simpl. repeat split; auto.
  Qed.
End Command.
```

D.4 Total.v

```
Require Import TLA.
Require Import Rules.
Require Import Arith.
Require Import ZArith.
Require Import Omega.
Require Import Setoid.
Require Import FunctionalExtensionality.
Require Import Memory.
Require Import Command.
Require Import Rules.
Module Total (Params : MemParams).
  Module MEMORY := Memory (Params).
  Module COMMAND := Command (MEMORY.FROMMEM).
  Print MEMORY.BASIC.St.
  Print COMMAND.BASIC.St.
  Definition castSt1 (st : MEMORY.BASIC.St) : COMMAND.BASIC.St :=
    match st with
    | {| MEMORY.BASIC.ready := ready;
           MEMORY.BASIC.status := status;
           MEMORY.BASIC.memory := memory;
           MEMORY.BASIC.output := output |} ⇒
             {|
               COMMAND.BASIC.ready := ready;
               COMMAND.BASIC.status := status;
               COMMAND.BASIC.memory := memory;
               COMMAND.BASIC.output := output |}
    end.
  Definition castSt2 (st : COMMAND.BASIC.St) : MEMORY.BASIC.St :=
    match st with
    | {| COMMAND.BASIC.ready := ready;
           COMMAND.BASIC.status := status;
           COMMAND.BASIC.memory := memory;
           COMMAND.BASIC.output := output |} ⇒
             {|
               MEMORY.BASIC.ready := ready;
               MEMORY.BASIC.status := status;
               MEMORY.BASIC.memory := memory;
               MEMORY.BASIC.output := output |}
    end.
  Lemma castStInv1 : ∀ st, st = castSt2 (castSt1 st).
  Proof.
    intros st. destruct st; auto.
  Qed.

  Lemma castStInv2 : ∀ st, st = castSt1 (castSt2 st).
  Proof.
    intros st. destruct st; auto.
  Qed.

  Definition sameSt (st1 : MEMORY.BASIC.St) (st2 : COMMAND.BASIC.St) :
Prop :=
    castSt1 st1 = st2.

  Definition sameStAlt : ∀ st1 st2, sameSt st1 st2 ↔ st1 = castSt2 (st2).
  Proof.
    intros st1 st2. unfold sameSt.
    destruct st1, st2. split; intros H; inversion H; auto.
  Qed.

  Inductive St : Type := Make_St : ∀ st1 st2,
    sameSt (MEMORY.refinement_mapping st1) (COMMAND.refinement_mapping st2)
→ St.

  Definition p1 (st : St) : MEMORY.St :=
    match st with
    | Make_St m _ _ ⇒ m
    end.

  Definition p2 (st : St) : COMMAND.St :=
    match st with
    | Make_St _ m _ ⇒ m
    end.

  Definition Spec : Expr St := (map p1 MEMORY.Spec) ‘∧ (map p2 COMMAND.Spec).

End Total.
```