SAND2016-4322C

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, Geoffrey C. Hulette, and Zena M Ariola

No Institute Given

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 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. Not applying formal methods upfront does not eliminate the need for verification and so often formal methods are applied after the fact. Not surprisingly, post hoc verification turns out to be very difficult in practice–in some cases work is ongoing on verification problems for systems first deployed 30 years ago. Part of the reason why post hoc verification is so difficult is that "correctness" of a system is always actually correctness with respect to some specification of the system. Thus, in order to show high level correctness properties of high consequence systems those systems must have tractable and accurate formal specifications. Faced with this reality, Sandia engineers are implementing new methodology that integrates formal methods into the specification and design phase of high consequence systems. This makes particular demands on the methods used. 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, however, we need tools which make verification tractable at scale.

Sandia engineers have begun to make use of the Temporal Logic of Actions [12] for formalizing specifications. TLA has proven itself to be an effective formalism for describing the evolution of digital systems over time. Key to its effectiveness is that the temporal aspects of the logic are available while needed but do not overwhelm the user when they are not. Generally much of the specification of a digital system is not directly temporal [14]. Lamport's TLA+ handles the non-temporal aspects of verification via an associated mathematics language which closely follows ordinary set theory. Specifications can therefore be constructed using standard mathematical techniques. Using the TLA+ tools it is possible to construct and verify complex system specifications using the specialized support for temporal reasoning for temporal properties the more general mathematical functionality for non temporal ones.

However, TLA+ and its associated tooling still faces limitations which impact its use in development of large scale high consequence systems. TLA+ comes with a model checker [21], but this model checker, while fully automated, is insufficient for proving many properties. Often times engineers need to work over infinite, or at the very least very large, state spaces and want to verify properties beyond the capability of any model checker. For such properties fully automated methods may be insufficient, requiring instead interactive theorem proving with human supplied lemmas and inductive hypotheses. What is more, model checkers are complex pieces of software whose correctness may not be self evident. For highly critical systems we want to be able to independently verify claims made by both automated and interactive methods. 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 with TLA+ we have embedded a version of TLA inside the proof assistant Coq [7]. Coq is a highly advanced tool which overcomes many of the limitations of TLA+, however, unlike TLA its logic is not particularly suited 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}.

An initial problem which motivated the design of TLA^{Coq} is the design and specification of a real digital component of a high consequence digital system being produced by Sandia. We initially developed a formal model of this component, called the Arbitrary Waveform Generator (AWG) in TLA+ but after running into limitation in TLA+ we developed TLA^{Coq} and transitioned the AWG model over to our embedding. Doing so allowed us to prove properties which we could otherwise only partially verify by way of bounded model checking and 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 more widely and to bigger systems.

The remainder of this paper is structured as follows. 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.

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 AWGs original requirements document that otherwise might have been missed while 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 safety critical and non safety critical systems. 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 the control system for an aircraft actuator. Given both the generality of the AWG and its importance in a concrete high consequence system we believe it serves as useful 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 upon receiving a special signal to 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

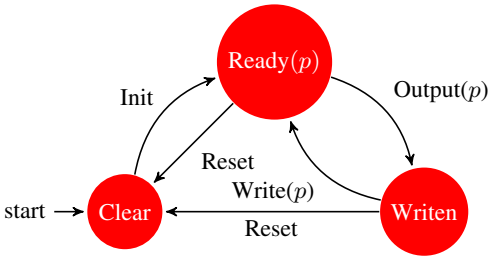


Fig. 1. Simplified AWG state machine

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 something like the initial state.

Of course, this simply state machine description is not complete. We also need to describe the input and output protocols by which the AWG interacts with its environment as in Figure 2.

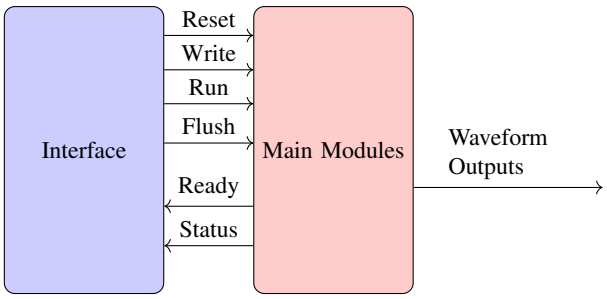


Fig. 2. AWG logical interface

This description of the AWG is, of course, much more abstract than what we 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 a 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 want to refine protocols in addition to state machines. And, this is necessary to build tractable high level specifications.

Thus we will refine our AWG model to clarify that the pattern is played out not all at once but as a sequence of states requiring an internal memory inside the component. Additionally we need to refine our AWG model to describe the channels on which it communicates. 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 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 formulate the specification for the AWG system we turned to the Temporal Logic of Actions (TLA). TLA has desirable properties for formulating a system like the AWG. Crucially it is stuttering invariant meaning, intuitively, that TLA specifications are not sensitive to the rate of the passage of time [3]. This enables the development of formal TLA specification using a process of refining more abstract specification into more concrete ones in much the same way think about designs informally [5]. TLA takes a logic centric view: specification and theorems about specification are both just logical formula 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 means 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 An Abstract Model

We began formulating the specification of the AWG using the TLA+ toolset via a process of refinement. Our initial specification, therefore, only modeled some very basic properties of the AWG system. However, despite deferring many details to later refinements the basic model is still reasonably complex. This model is parameterized by a set called `Pattern` which serves as as to abstract away the details of the patterns stored in memory and played out by the AWG. We also assume the existence of a constant `NilPattern` and require that `NilPattern` be an element of `Pattern`. A special set `MemStatus` is defined as containing only the constants `Valid` and `Invalid`. The states of our specification are then described using four variables consisting of the `memory` together with the three output lines `ready`, `status`, and `output`. The type invariant which we will prove about the system states that `ready` is always a boolean, `status` is always in `MemStatus`, and that both `output` and `memory` are always elements of `Patterns`.

The `Next` relation is then defined as the conjunction of five actions,

```
Next ==
  \/ Initialize
  \/ Reset
  \/ Flush
  \/ \E p \in Pattern : Write(p)
  \/ Run
```

one of which `Write` is parameterized by a member of `MemStatus`. `Write`(p) is possible whenever the status is `Valid` and ready is true. Its effect is to set the memory to the value of p unless p is the `NilPattern` in which case the memory is left unchanged. All other variables are also unchanged.

```
Write(p) ==
  /\ ready = TRUE
  /\ status = Valid
  /\ IF p /= NilPattern
     THEN memory' = p
     ELSE UNCHANGED memory
  /\ UNCHANGED <<ready, status, output>>
```

`Run` can also occur whenever the `status` is `Valid` and `ready` is `true`. `Run` has the effect of setting the new value for output to the initial value of memory and setting the new value of memory to `NilPattern` unless the value of memory was already `NilPattern` in which case neither memory nor output are changed. All other variables are left the same.

```
Run ==
  /\ ready = TRUE
  /\ status = Valid
  /\ IF memory /= NilPattern
     THEN
        /\ output' = memory
        /\ memory' = NilPattern
     ELSE UNCHANGED <<output, memory>>
  /\ UNCHANGED <<ready, status>>
```

Similar to `Run` and `Write`, `Flush` is enabled whenever the `status` is `Valid` and `ready` is `true`. Its effect is to simply set both `memory` and `output` to the `NilPattern` leaving `status` and the `ready` flag unchanged. By contrast, `Reset` has no preconditions and is thus always enabled. `Reset` must set the `ready` flag to `false` and the output to the `NilPattern`, but can otherwise set the remaining variables to any type correct values.

```
Reset ==
  /\ ready' = FALSE
  /\ status' \in MemStatus
  /\ memory' \in Pattern
  /\ output' = NilPattern
```

Finally, the `Initialize` action is enabled whenever `ready` is `false`. Its effect is to set ready to true and to set the `status` flag to one of the values in `MemStatus`. If the new `status` is `Valid` then `Initialize` will also set the `memory` to `NilPattern` and leave the `output` unchanged. Otherwise both `memory` and `output` are left unchanged.

At `PowerOn` our basic model sets the `ready` flag to `false` and the output to the `NilPattern` but any type correct values are permitted for the other variables.

The `Initialize` action plays a crucial role in this model is it is what takes us from an unready state after a reset or on powering on to a ready one. As such, we additionally require that `Initialize` be handled fairly.

```
Fairness == WF_vars(Initialize)
```

The weak fairness assumption ($WF(A)$) in TLA simply demands that either A keeps happening (that is, it always eventually happens) or it is blocked infinitely often (always eventually impossible) [14]. Our full specification for the basic model then is the conjunction of the system starting in a `PowerOn` state, it advancing by either stuttering or carrying out the next action, and the fairness condition.

```
Spec == PowerOn /\ [][Next]_vars /\ Fairness
```

We can state a number of important properties about this specification. Perhaps most important is the type safety property: the specification implies that the variables always take values from the appropriate sets.

```
THEOREM Spec => [] TypeInvariant
```

There are other properties of importance, however, like that `Reset` is always enabled or that the `Flush`, `Run`, and `Write` commands are enabled exactly when the the ready flag is true and the status code is valid.

```
CommandsEnabled ==
  /\ ENABLED Run
  /\ ENABLED Flush
  /\ \A p \in Pattern : ENABLED Write(p)

CommandsEnabledIffValid ==
  CommandsEnabled <=> (ready = TRUE /\ status = Valid)

THEOREM Spec => [] CommandsEnabledIffValid
THEOREM Spec => [] (ENABLED Reset)
```

So far we have only looked at safety properties, but we also care about liveness properties. Most importantly is the idea that the system can never have a `false ready` flag forever

```
THEOREM Spec => (ready = FALSE ~> ready = TRUE)
```

3.2 A Refinement

The basic model does not incorporate all the relevant details of the AWG specification. We thus need to refine the model to a more detailed specification. One of the areas of missing details 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 model where we incorporate commands our state consists of the four variables from 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. The set `Command` is defined as consisting of the constants `Flush`, `Run` and the constant `Write` together with a payload containing a `Pattern`. We further define the set `Control` to consist of the constants `Ready`,`Busy`, and `Done`. The type invariant is extended with two new clauses indicating that `ctrl` is from the the set `Control` and that cmd is from the set `Command`.

The `Reset` action from basic is preserved only extended to set `ctrl` and `cmd` to arbitrary values which respect the type invariant. The `Initialize` event is similarly preserved except that now it sets `ctrl` to `Ready` and leaves `cmd` unchanged. However, in place of the actions `Flush`, `Run`, and `Write`(p) we have a more complicated implementation of commands by way of the actions `Do`, `Resp`, and `Req`(c).

```
Next ==
  \/ Initialize
  \/ Reset
  \/ \E c \in Command : Req(c)
  \/ Do
  \/ Resp
```

The action `Resp` can occur whenever the `ready` flag is `true`, the `status` flag is `Valid`, and the `ctrl` variable is `Done`. Its only effect is to advance `ctrl` to `Ready` andto leave everything else unchanged. The parameterized action `Req`(c) is used for reading commands off the input. It can happen when `ready` is `true`, the `status` is `Valid`, and `ctrl` is `Ready`. It sets `cmd` to the value c and `ctrl` to `Busy` but leaves the other variables unchanged.

The action `DoFlush` is enabled whenever `cmd` is `Flush` and has the effect of setting both `memory` and `output` to `NilPattern`. The action `DoRun` is enabled whenever `cmd` is `Run` and has the effect of setting the new value of `output` to the prior value of `memory` and setting the new value of `memory` to `NilPattern`. The action `DoWrite` is enabled whenever `cmd` is `Write`(c) for some value of c and has the effect of setting `memory` to c unless this c is the `NilPattern` in which case c is left unchanged. In either case `DoWrite` leaves the `output` unchanged. These three actions are disjunctively combined into the `Do` action which additionally requires that the `status` reads as `Valid` and that the `ready` flag reads is `true` and, crucially, that `ctrl` is set to `Busy`. After execution of `Do`, `ctrl` is set to `Done` while the `ready`, `status`, and `cmd` variables are left unchanged.

The more complicated account of the implementation of commands also requires additional fairness assumptions. It would be a problem if the system just stuttered indefinitely ready to perform a command but did not ever actually perform one. Thus, we require weak fairness hold for both `Resp` and `Do`.

There are a number of properties we expect to follow from the specification. As before we want the specification to imply that the type invariant always holds. We also want to ensure that what we have described is a refinement, which is to say we want the command version of `Spec` to imply the `Spec` from the basic version. There are also additional liveness properties that we are interested in. Chief among these is the completeness of the implementation of commands: if at any point `ctrl` is `Busy` then at some later point `ctrl` will be `Ready`.

3.3 Another Refinement

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 empty sequence and simply switches the `mode` back to `Record`.

3.4 Limitations of the TLA framework

At this point we have described a rather extensive set of models for the AWG in TLA. 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 care about. 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 somehow the refinement of the basic model which extends it 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.

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

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 and 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 [20] come equipped with fully powerful fully automated methods (such as Coq's Presburger solver which uses the Omega test [16]) as well as tools for developing new domain specific automation, but these capabilities are kept separate from the logic kernel. Curry-Howard based systems further unify the language of terms which the logic is about with the language of proofs, allowing proofs to be treated as first class objects and worked with directly in the system. By being fundamentally based on computation, type theory based systems are also intrinsically geared to verifying algorithmic systems. Powerful type systems like Martin-Löf Type Theory [15] or the Calculus of Constructions [6] are rich enough not just to formalize mathematics but to build it directly as an alternative foundation to set theory. Indeed, born of its lineage in constructive mathematics type theory makes a stronger claim than that: type theory isn't just an alternative to set theory, it is potentially superior as the structural approach of type theory in which types of mathematical objects (such as pairs or lists) are defined directly by their formal properties rather than being encoded into sets more closely matches intuition and everyday mathematical practice.

Coq in particular seems like a close match to our needs. It's 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 [11]. However, unlike what we see with TLA+, Coq's logic is not geared toward expressing systems which evolve over time. While TLA is well suited to describing systems like the AWG, vanilla Coq is not.

TLA^{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 [13, 14]. The Coq code in Figure 3 provides a

```
Require Import TLA.
Require Import ZArith.
Require Import Omega.

Local Open Scope Z.
Local Open Scope tla.

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

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. While exceedingly simple, the clock already demonstrates the main features of our embedding 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 will always be between one and twelve. The proof in Figure 4 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 formula 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 (the `intuition` tactic) becomes sufficient

```
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. 4. Type Invariant for the Clock in Coq

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 mathematical library. The availability of theorems such as these is one of the benefits of working with a system such as Coq.

Using TLA^{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. Coq has multiple features allowing us to abstract over a type but in this case the approach we took was to use Coq's module language.

```
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.
```

We describe the parameters to the basic module by way of a module type BasicParams which has a type representing patterns. In order that we might later instantiate this type with something like the type of sets which would be larger than the set of parameters, we call this type `PatternIsh` and also include a parameter `IsPattern` which is a predicate on `PatternIsh`. The next parameter is the `NilPattern`, a member of `PatternIsh`. We then require two axioms: `NilPattern` must be a pattern, and every pattern must be either equal to the `NilPattern` or different from it. This second condition holds automatically in a classical setting, but as Coq uses intuitionistic logic unless we explicitly assume classical principles, it must be stated explicitly in our case. Indeed, as we expect `PatternIsh` to be eventually instantiated with a type with de-

cidable equality (such as the type of n-arry bit vectors), assuming only that equality is classical and not that it is decidable is actually quite conservative.

The basic module then technically takes the form a functor, parameterized by a parameter module of the basic module type.

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

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`.

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

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

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.

```
Definition Next :=
 Initialize
  \/ Reset
  \/ Flush
  \/ (exists p, IsPattern p /\ Write p)
  \/ Run.
```

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.

```
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).
```

This boilerplate is an unfortunate consequence of encoding into Coq, but is only a constant factor larger than what we encounter in TLA+ where we must list all variables explicitly at the top of a section of code. 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.

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

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. That `status` is in the type `MemoryStatus` or that `ready` is a boolean are not theorems about the specification but part of the type of states of which the specification is about. However, because we encode membership in the pattern set as a predicate, we still have a type invariant that simply states that output and memory remain valid patterns.

```
Definition TypeInvariant :=
  IsPattern memory /\ IsPattern output.
```

However, by switching to the Coq representation we can not *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 basic. It is again constructed as a module functor parameterized by a module of the `BasicParams` type.

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

Using Coq's inductive type system makes it easy to express the type of the control variable as well as the type of commands.

```
Inductive Control : Type := Ready | Busy | Done.
Inductive Command : Type :=
  | Flush : Command
  | Write : PatternIsh -> Command
  | Run : Command.
```

However, 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.

```
Definition IsCommand (c : Command) : Prop :=
  match c with
    | Write p => IsPattern p
    | _ => True
  end.
```

Then, the states of the command module are expressed by way of a record type with six fields.

```
Record St := {
      ready : bool;
      status : MemStatus;
      ctrl : Control;
```

```
    cmd : Command;
    memory : PatternIsh;
    output : PatternIsh}.
```

While, the type invariant property must now also include that command given is well formed.

```
Definition TypeInvariant :=
    IsCommand cmd
    /\ IsPattern memory
    /\ IsPattern output.
```

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. Except, now, we prove our various theorems in Coq. However, because we can not simply take advantage of variable punning the way TLA+ does to treat states from the command module as states of the basic module we must express the refinement by way of a refinement mapping function after first instantiating the basic module.

```
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.
Theorem refines :
    valid (Spec '=> (map refinement_mapping BASIC.Spec)).
```

The last module to translate is the memory module. Here we have 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`.

```
Module Type MemParams <: BasicParams.
  (* some code elided *)
  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.
```

The main idea then is to instantiate the basic module with a representation in which patterns from the basic correspond to sequences of patterns in the memory module. We do this by way of a module functor.

```
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.
```

Memory itself is simply parameterized by a module of type `MemParams` and has its states a record with five fields.

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

As before the Coq code is mostly a direct translation of the original TLA+ specification. Only now we can give Coq checked proofs of properties rather than depending on model checking. The refinement mapping from Memory to Basic, however, depends on instantiating Basic with a *different*, and more specific, set of parameters.

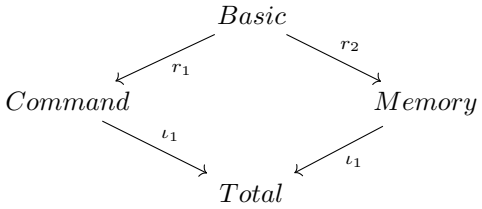
```
Module FROMMEM := ParamsFromMem (Params).
Module BASIC := Basic.Basic (FROMMEM).
Definition refinement_mapping (st : St) : BASIC.St :=
    match st with
      | {|  ready := ready0;
            status := status0;
            memory := memory0;
            mode := mode0;
            output := output0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

```
        {|  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.
```

Now, we have constructed two modules in Coq which each define a TLA$^{\text{Coq}}$ specification refining our original basic specification. Our next goal then is to define 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.

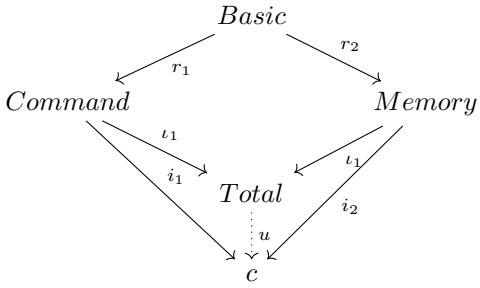
One option, available to us from TLA, is to simply combine the two models as a conjunction [2]. However, a complexity arises in that the two models have different type to represent their states and we need to be explicit about what states we are using. Using the product of the two state spaces would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

Let us therefore be specific about what we want. 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 be familiar to those who have encounter category theory as it is what is called, in categorical language, a "push-out." Indeed, more than 20 years ago, Goguen suggested push-outs as an abstract, formalism independent, definition for the combination of components [10]. And, Fiadeiro and Maibaum already demonstrated the use of push-outs for combining specifications in a temporal logic setting in the early 1990s [8]. 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.

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).
```

5 Implementing TLA^{Coq}

There are a number of ways to use a meta logic as rich as Coq to host another logic or language. Often times when considering embedded languages we contrast so called "shallow" and "deep" embeddings [18]. In a shallow embedding each construct of the target language is mapped to a function in the meta language which directly implements that construct. By contrast, in a deep embedding the target language constructs are considered as data representing syntax. Existing embeddings of TLA into Coq have taken both of these approaches. In [19] a shallow embedding of Coq in TLA was constructed to prove a meta result that all TLA specifications of a certain form satisfy a "machine closed" property 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 [17]. 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. They, following, Lamport, use a two step approach where their logical syntax corresponds to a full set of "RTLA" formulae of which the stuttering invariant TLA formula 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 inductively defined datatype with constructors for the logical operators like conjunction, disjunction, and implication as well as the temporal operators such as eventually and forever. This data type, given in Figure 5 is parameterized by a type `A` which is the type of states. Crucially though, this representation ensures that all formula are stuttering invariant by construction. This is important to us because it later allows us to use an induction principle for arbitrary safety properties (`tla_inv_gen` discussed below) which does not generate any proof obligations relating to stuttering steps. The desire to give such induction principle motivates our slightly different design decisions from prior work. Because formulae are rep-

```
Variable A : Type.

Definition Predicate := A -> Prop.
Definition Action := A -> A -> Prop.

Inductive Expr :=
| E_Predicate : Predicate -> Expr
| E_AlwaysAction : forall {B : Type},
    (A -> B) -> Action -> Expr
| E_EventuallyAction : forall {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 : forall {B : Type}, (B -> Expr) -> Expr
| E_ExistsRigid : forall {B : Type}, (B -> Expr) -> Expr.

Definition E_Enabled (r : Action) :=
  E_Predicate (fun st => exists st', r st st').
```

Fig. 5. Inductive type representing TLA expressions

resented as an inductive data type it becomes possible to define functions which work with these formulae using recursion. Moreover, we can prove meta logical properties about formulae in general using induction. However, in the case of the leaves of TLA formulae–predicates about a current state and actions relating two states–our encoding

switches to simply using Coq propositions. That is, when it comes to the mathematical language used in formulae we simply use Coq directly rather than continuing to use a deep embedding to encode some powerful logic such as set theory. Similarly, we encode the quantifiers using functions whose codomain is expressions and whose domain is some Coq type.

On top of this data type we use CoqÕs notations facilities to build a more convenient syntax for writing TLA formulae. Our general convention is to begin TLA syntactic constructions with a back tick, as in case of conjunction.

```
Notation "P `/\ Q" :=
 (E_Conj P Q) (at level 80, right associativity): tla_scope.
```

Formulae however are only have the picture. We don't just want to write formulae: we want to prove them true. We are thus faced with a problem of saying what exactly qualifies as a proof of a formula. One option would be to construct a syntactic proof theory for TLA as a Coq data type and then to encode proofs as objects in this datatype. However, we chose instead to switch semantical account of truth of TLA formulae in Coq and then to use Coq to prove the truth of a formula. This is essentially the same approach taken by Lamport: we define TLA in terms of its semantics rather than in terms of its proof theory [12]. TLA formulae are interpreted as being predicates over behaviors where behaviors are infinite sequences of states. We encode such infinite sequences simply as functions from natural numbers to states. We define the operator @ taking a behavior and a natural number such that $s@n$ is the behavior created by forgetting the first n states of s.

```
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 =>
        forall n, (r (s@n 0) (s@n 1)) \/ f (s@n 0) = f (s@n 1)
    | E_EventuallyAction f r => fun s =>
        exists 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 => forall n, eval e (s@n)
    | <> e => fun s => exists n, eval e (s@n)
    | E_ForallRigid f => fun s => forall x, eval (f x) s
    | E_ExistsRigid f => fun s => exists x, eval (f x) s
  end.
```

Fig. 6. Semantics of TLA in Coq

The interpretation of a formula then is a function from behaviors to propositions given by the coq function `eval` given in Figure 6. The idea is that a formula E with is true of a behavior S if `eval` E S is. From `eval` we can define the notion of validity of a formula as being that it is true of every behavior.

```
Definition valid (A : Type) (e : Expr A) : Prop :=
  forall s, eval e s.
```

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) [12]. 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.
```

Fig. 7. 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 7). We can then show that stuttering equivalence is indeed an equivalence relation and that it is a stuttering relation.

With the definition of stuttering equivalence in place we can prove the metatheorem that the truth value of a formula is always stuttering invariant.

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

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

```
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).
```

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 mapping a state change function over a formula (given in Figure 8). 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

```
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. 8. Mapping over an expression to change the state type

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 [9]. 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^{Coq} specifications from these models. Using the type of sets may be helpful in this situation.

6 Conclusions

By embedding TLA in a foundational proof assistant like Coq we were able to construct a model of a critical system component in a compositional manner using refinement. Not only did this approach allow us to express the specification we cared about, but we could construct proofs of correctness properties using an interactive theorem proving approach combining automation with human creativity to come up with intermediate

states like induction hypotheses. Because these proofs occur within Coq's logic we achieve a high level of confidence in our proofs: we do not depend on the correctness of a temporal logic specific proof engine or external verification tools, only on the Coq kernel and its implementation of Coq's reasonably simple logic.

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.

Our use of Coq allowed us to actually prove all our stated properties rather than depend on bounded model checking. However, the use of proofs rather than model checking is not without cost. Model checking is a full automated method while our inductive proofs require user input. Therefore, an important topic for future work is improving the automation available within the Coq embedding. There are a number of paths to improve the quality of automation, but at an obvious one is to simply replicate the capabilities of the TLA+ model checker within Coq. It should be possible to implement a model checker for TLACoq in Coq and, even, potentially, to prove its correctness. We believe that many of the properties we proved "by hand" could be shown more or less automatically by such a system. In other cases, full model checking will not be tractable, but having a tool for bounded model checking within TLACoq would be useful when developing models as it would provide designers and verifiers a low cost way to check that they are on the right track.

Coq already serves as a common framework for formal reasoning at many levels, ranging from abstract mathematical proof to the development of certified compiler. Currently, we are using TLA and TLACoq to prove properties about models of high consequence systems, however, at some point these systems are implemented in hardware or low level languages and those implementations are not certified.. Another important direction for future work therefore is to link up our work on TLACoq with other Coq based verification approaches such that eventually we can get to the point of "full stack verification" where low level implementations are proven to correspond to our high level models. This would take us towards our eventual goal of a development approach where design, development, and specification happen in tandem leading to eventually implementations which are fully verified and correct by construction, allowing us to assert with certainty that abstract safety properties hold about the concrete complicated digital systems we deploy in high consequence applications. Formalizing the high level descriptions of systems as we have done with TLACoq 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. COHEN, E., AND LAMPORT, L. Reduction in tla. In *CONCUR'98 Concurrency Theory*, D. Sangiorgi and R. de Simone, Eds., vol. 1466 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 1998, pp. 317–331.
6. COQUAND, T., AND HUET, G. The calculus of constructions. *Information and Computation 76*, 2 (1988), 95 – 120.
7. DEVELOPMENT TEAM, T. C. *The Coq proof assistant reference manual*. LogiCal Project, 2004. Version 8.0.
8. FIADEIRO, J., AND MAIBAUM, T. Temporal theories as modularisation units for concurrent system specification. *Formal Aspects of Computing 4*, 3 (1992), 239–272.
9. GAMBINO, N., AND ACZEL, P. The generalised type-theoretic interpretation of constructive set theory. *Journal of Symbolic Logic 71* (3 2006), 67–103.
10. GOGUEN, J. A. A categorical manifesto. *Mathematical structures in computer science 1*, 01 (1991), 49–67.
11. 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.
12. LAMPORT, L. The temporal logic of actions. *ACM Trans. Program. Lang. Syst. 16*, 3 (May 1994), 872–923.
13. LAMPORT, L. Refinement in state-based formalisms. Tech. rep., DEC Systems Research Center, 1996.
14. LAMPORT, L. *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.
15. NORDSTRÖM, B., PETERSSON, K., AND SMITH, J. M. *Programming in Martin-Löf's type theory, volume 7 of International Series of Monographs on Computer Science*. Clarendon Press, Oxford, 1990.
16. 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.
17. 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.
18. SVENNINGSSON, J., AND AXELSSON, E. *Trends in Functional Programming: 13th International Symposium, TFP 2012, St. Andrews, UK, June 12-14, 2012, Revised Selected Papers*. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013, ch. Combining Deep and Shallow Embedding for EDSL, pp. 21–36.

19. WAN, H., HE, A., YOU, Z., AND ZHAO, X. Formal proof of a machine closed theorem in coq. *Journal of Applied Mathematics* (2014).
20. WENZEL, M. *The Isabelle/Isar Reference Manual*, 2012.
21. YU, Y., MANOLIOS, P., AND LAMPORT, L. Model checking tla+ specifications. In *Correct Hardware Design and Verification Methods* (1999), Springer-Verlag, pp. 54–66.