CHP State and Trace Semantics Equivalence: A Coq Proof Guide

Stephen Longfield, Brittany Nkounkou, Rajit Manohar, and Ross Tate

last updated on April 16, 2015

This document serves as a guide to the Coq proof appended to our paper, “Preventing Glitches and Short Circuits in High-Level Self-Timed Chip Specifications”. The proof serves as mechanically verified evidence of our theorems and corollary stated in **Section 3.3**. The final corollary says that given the state semantics defined in **Figures 4, 5** and the trace semantics defined in **Figures 6, 7**, a CHP program is erroneous in the state semantics if and only if it is erroneous in the trace semantics. More precisely, for all CHP programs P not containing data-carrying channels and not containing $A¡$ or $A¿$ for any Channel A:

$$P \xrightarrow{s} \textbf{error} \;\Leftrightarrow\; P \xrightarrow{t^*} \textbf{error}$$

The final Coq construct that completes the proof of this statement can be found in `ProofFinal.v`, where we provide the definition of `BigErrorStep_iff_BigErrorTrace`, appropriately of type `forall P : Program, initial P -> (BigErrorStep P <-> BigErrorTrace P)`. It is later made clear that this Coq type corresponds exactly to the statement above. In this document, we generally use the term “step” to denote a single step in the state semantics, and the term “trace” to denote a single step in the open-world trace semantics.

We begin this guide by providing a visual overview of our Coq files, including a high-level description of each file and the logical dependencies between them. The logical dependencies appropriately indicate the order in which the files must be compiled in Coq. We verified our files with Coq version 8.4pl3. Following the visual overview, we highlight the important constructs within each file. Note that code excerpts have been modified for simplicity and brevity. For more details, the reader should refer to the full Coq code itself, which includes some guiding comments.

Visual Overview of Coq files

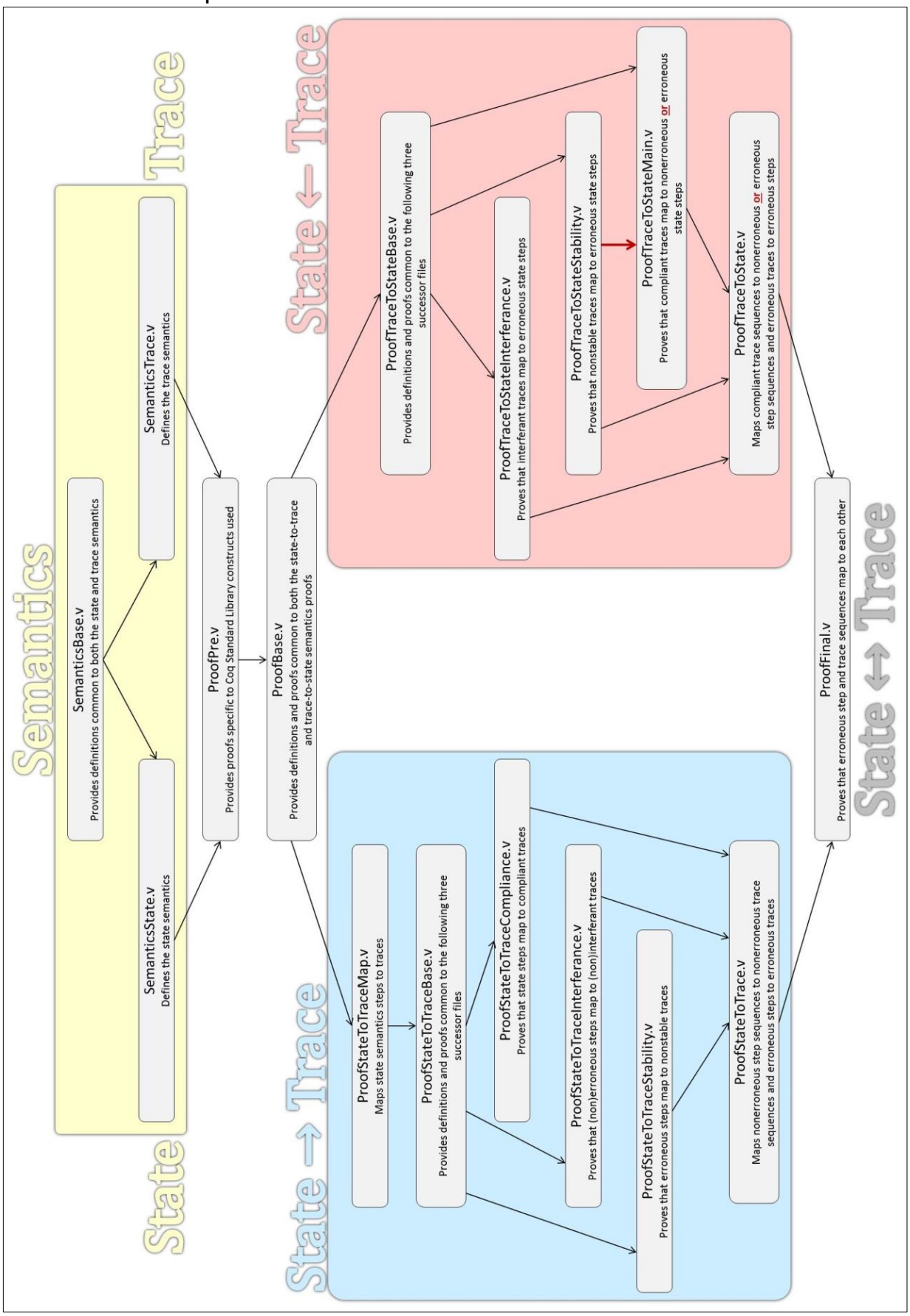

Figure 1: A visual overview of the complete Coq proof system, including logical dependencies between files

Table of Contents

SemanticsBase.v

In correspondence with **Figure 3** and Communication's extension in **Figure 4**, this file provides definitions common to both the state and trace semantics.

We assume a Channel type as follows:

```
Variable Channel : Type.
```

This type, along with decidable equality within the type, is the only assumption made throughout the entire proof system.

The Communication type is defined as follows:

```
Inductive Communication : Type :=
  | bang  (A : Channel) : Communication
  | bang' (A : Channel) : Communication
  | quer  (A : Channel) : Communication
  | quer' (A : Channel) : Communication.
```

where `bang A`, `bang' A`, `quer A` and `quer' A` represent $A!$, $A¡$, $A?$ and $A¿$, respectively.

The Guard type is defined as follows:

```
Definition Guard : Type := list Channel.
```

where the list `[A ; … ; A']` represents the Boolean expression $\bar{A} \wedge \ldots \wedge \bar{A'}$.

The Program type is defined as follows:

```
Inductive Program : Type :=
  | skip                              : Program
  | comm (C : Communication)          : Program
  | sequ (P0 P1 : Program)            : Program
  | par  (P0 P1 : Program)            : Program
  | rep  (P : Program)                : Program
  | det  (GP0 GP1 : Guard * Program)  : Program
  | ndet (GP0 GP1 : Guard * Program)  : Program.
```

where the constructors above intuitively represent skip, a Communication, sequential composition, parallel composition, infinite repetition, deterministic choice and nondeterministic choice.

Finally, the Probe type is defined as follows:

```
Inductive Probe : Type :=
  | bar (A : Channel) : Probe
  | hat (A : Channel) : Probe.
```

where `bar A` and `hat A` represent the variables $\bar{A}$ and $\hat{A}$, respectively. Note that a "Probe" as defined here does not appear in Programs. Instead, it is used to express states of the state semantics and classification functions of the closed-world trace semantics. Following the definition of the Probe type, we additionally prove some statements about Probe equality.

SemanticsState.v

This file defines the state semantics in correspondence with **Figure 5** and the unchanged rules from **Figure 4**.

We first define the State type as follows:

```
Definition State : Type := list Probe.
```

where the number of a Probe's occurrences in the list corresponds to the natural number to which the State maps that Probe. With this, we further define State equality, the initial State, and a State's evaluation of a Guard as follows:

```
Definition eq_State (s1 s2 : State) : Prop :=
  forall p, count_occ s1 p = count_occ s2 p.

Definition state_init : State := nil.

Fixpoint s1 (s : State) (G : Guard) : Prop :=
  match G with
    | nil => True
    | A :: G' => count_occ s (bar A) > 0 /\ s1 s G'
  end.
```

Next, we define non-erroneous steps (SmallStep) and sequences of them (BigStep) as follows:

```
SmallStep : Program -> State -> Program -> State -> Type := …

Inductive BigStep : Program -> State -> Program -> State -> Type :=
  | S' : BigStep P s P s
  | SS : SmallStep P s P' s' -> BigStep P s P' s'
  | BS : BigStep P s P' s' -> eq_State s' s''
      -> BigStep P' s'' P'' s''' -> BigStep P s P'' s'''.
```

Then, we define erroneous steps (ErrorStep) as follows:

```
Nonstable : Program -> State -> State -> Type := …

Inductive ErrorStep : Program -> State -> Type :=
  | CommB2_E (c1 : In (hat A) s) (c2 : count_occ s (bar A) > 1)
      : ErrorStep (comm (bang' A)) s
  | CommQ2_E (c1 : ~ In (bar A) s) (c2 : count_occ s (hat A) > 1)
      : ErrorStep (comm (quer' A)) s
  | Sequ_E (es : ErrorStep P1 s)
      : ErrorStep (sequ P1 P2) s
  | ParL_E (es : ErrorStep P1 s)
      : ErrorStep (par P1 P2) s
  | ParR_E (es : ErrorStep P2 s)
      : ErrorStep (par P1 P2) s
  | Det2_E (c1 : s1 s (fst GP0)) (c2 : s1 s (fst GP1))
      : ErrorStep (det GP0 GP1) s
  | ParL_NSE (ss : SmallStep P1 s P1' s')
             (ns : Nonstable P2 s s')
      : ErrorStep (par P1 P2) s
  | ParR_NSE (ss : SmallStep P2 s P2' s')
             (ns : Nonstable P1 s s')
      : ErrorStep (par P1 P2) s.
```

where Nonstable corresponds to the last rule in **Figure 5**.

Finally, we define an erroneous step sequence (BigErrorStep) as follows:

```
Inductive BigErrorStep : Program -> Type :=
```

```
  | BES (bs : BigStep P state_init P' s') (es : ErrorStep P' s')
      : BigErrorStep P.
```

SemanticsTrace.v

This file defines the trace semantics in correspondence with **Figure 6** and **Figure 7**.

We first define the open-world trace semantics as follows:

```
Inductive Move : Type :=
  | start  : Move
  | use    : Move
  | finish : Move.

Inductive Step : Type :=
  | send (M : Move) (A : Channel) : Step
  | recv (M : Move) (A : Channel) : Step.

Inductive Label : Type :=
  | idle                    : Label
  | step   (S : Step)       : Label
  | holds  (G : Guard)      : Label
  | holds2 (G1 G2 : Guard) : Label
  | ll     (a0 a1 : Label) : Label.

Trace : Program -> Label -> Program -> Type := …
```

Then, we define the closed-world semantics, including non-erroneous trace sequences (BigTrace), single erroneous traces (ErrorTrace), and erroneous trace sequences (BigErrorTrace), as follows:

```
Fixpoint pre (a : Label) : list Probe :=
  match a with
    | step (send use A) => (bar A) :: nil
    | step (recv use A) => (hat A) :: nil
    | step (send finish A) => (bar A) :: nil
    | step (recv finish A) => (hat A) :: nil
    | lr a0 a1 => (pre a0) ++ (pre a1)
    | _ => nil
  end.

Fixpoint post (a : Label) : list Probe :=
  match a with
    | step (send start A) => (bar A) :: nil
    | step (recv start A) => (hat A) :: nil
    | step (send use A) => (bar A) :: nil
    | step (recv use A) => (hat A) :: nil
    | lr a0 a1 => (post a0) ++ (post a1)
    | _ => nil
  end.

Fixpoint eval (G : Guard) (l : list Probe) : Prop :=
  match G with
    | nil => True
    | A :: G' => In (bar A) l /\ eval G' l
  end.
```

```
Fixpoint req (a : Label) (l : list Probe) : Prop :=
  match a with
    | idle => True
    | step (send start A) => ~ In (hat A) l
    | step (recv start A) => In (bar A) l
    | step (send use A) => True
    | step (recv use A) => True
    | step (send finish A) => In (hat A) l
    | step (recv finish A) => ~ In (bar A) l
    | holds G => eval G l
    | holds2 G1 G2 => eval G1 l /\ eval G2 l
    | lr a0 a1 => req a0 l /\ req a1 l
  end.

Fixpoint up (a : Label) : list Probe :=
  match a with
    | step (send start A) => (bar A) :: nil
    | step (recv start A) => (hat A) :: nil
    | step (send use A) => (bar A) :: nil
    | step (recv use A) => (hat A) :: nil
    | lr a0 a1 => (up a0) ++ (up a1)
    | _ => nil
  end.

Fixpoint down (a : Label) : list Probe :=
  match a with
    | step (send finish A) => (bar A) :: nil
    | step (recv finish A) => (hat A) :: nil
    | lr a0 a1 => (down a0) ++ (down a1)
    | _ => nil
  end.

Fixpoint detreq (a : Label) : Prop :=
  match a with
    | holds2 G1 G2 => False
    | ll a0 a1 => detreq a0 /\ detreq a1
    | _ => True
  end.

Definition compliant (a : Label) : Prop := req a (pre a).

Definition interferant (a : Label) : Type
  := {p : Probe & In p (up a) /\ In p (down a)}.

Definition stable (a : Label) : Prop := req a (post a).

Inductive BigTrace : Program -> Program -> Type :=
  | ST : Trace P a P' -> compliant a
         -> (interferant a -> False) -> BigTrace P P'
  | BT : BigTrace P P' -> BigTrace P' P'' -> BigTrace P P''.

Inductive ErrorTrace : Program -> Type :=
  | NDR_ET : Trace P a P' -> compliant a -> ~ detreq a -> ErrorTrace P
  | Int_ET : Trace P a P' -> compliant a -> interferant a -> ErrorTrace P
```

```
  | NSt_ET : Trace P a P' -> compliant a -> ~ stable a -> ErrorTrace P.

Inductive BigErrorTrace : Program -> Type :=
  | BET (bt : BigTrace P P') (et : ErrorTrace P') : BigErrorTrace P.
```

The closed-world trace semantics as they are constructed here appear to be quite different from the definition in **Figure 7**. This is primarily due to brevity and exposition concerns in the paper, where we expressed classification functions in terms of the simple separately defined **off** and **on** constructs, as opposed to using the composite construct req as is it appears here in the proof. Nevertheless, the two formulations are equivalent as follows (where boldface terms refer to constructs in the paper and italicized terms refer to the Coq constructs above):

1) $pre\ a$ and $post\ a$ correspond to the minimal states satisfying the respective columns of the chart in **Figure 7**. That is:
   a) $pre\ a \models \mathbf{pre}(a)$ and $\forall \sigma.\ \sigma \models \mathbf{pre}(a) \Rightarrow pre\ a \models P \Rightarrow \sigma \models P$
   b) $post\ a \models \mathbf{post}(a)$ and $\forall \sigma.\ \sigma \models \mathbf{post}(a) \Rightarrow post\ a \models P \Rightarrow \sigma \models P$
2) $req\ a\ l$, for some Label a and State l (remember a State is simply a list of Probes), corresponds to the statement $l \models \mathbf{off}(a) \wedge \mathbf{on}(a)$.
3) With this, $compliant\ a$ corresponds to $pre\ a \models \mathbf{off}(a) \wedge \mathbf{on}(a)$. It can be proven that this statement is equivalent to the definition of $\mathbf{compliant}(a)$ in **Figure 7** through use of 1a.
4) $stable\ a$ is dual to $compliant\ a$, where pre is replaced with $post$ (and 1b is used).
5) $up\ a$ and $down\ a$ are defined such that:
   a) $\forall p.\ p \in up\ a \Rightarrow \mathbf{drive}(a) \Rightarrow p$
   b) $\forall p.\ p \in down\ a \Rightarrow \mathbf{drive}(a) \Rightarrow \neg p$
6) By 5a and 5b, $\sim interferant\ a$ is equivalent to $\mathbf{noninterferant}(a)$.
7) The remaining Coq definitions above correspond directly to that in the paper. ($detreq$ corresponds to the last rule in **Figure 6**.)

It should be noted that the reasoning above requires that certain propositions be decidable, and we *can* claim that they are because we generally assume to be working in a finite environment (e.g. with a finite number of Channels, finite CHP Programs, etc.).

ProofPre.v

This file provides proofs specific to constructs defined in the Coq Standard Library that we used (e.g. lists). We refer the reader to the full Coq code for more details.

ProofBase.v

This file provides definitions and proofs common to both the state-to-trace and trace-to-state semantics proofs. Some notable definitions are:

```
Fixpoint initial (P : Program) : Prop :=
  match P with
    | skip => True
    | comm (bang A) => True
    | comm (bang' A) => False
    | comm (quer A) => True
    | comm (quer' A) => False
    | sequ P0 P1 => initial P0 /\ initial P1
    | par P0 P1 => initial P0 /\ initial P1
    | rep P' => initial P'
    | det GP0 GP1 => initial (snd GP0) /\ initial (snd GP1)
    | ndet GP0 GP1 => initial (snd GP0) /\ initial (snd GP1)
  end.
```

```
Fixpoint valid (P : Program) : Prop :=
  match P with
    | skip => True
    | comm _ => True
    | sequ P0 P1 => valid P0 /\ initial P1
    | par P0 P1 => valid P0 /\ valid P1
    | rep P' => initial P'
    | det GP0 GP1 => initial (snd GP0) /\ initial (snd GP1)
    | ndet GP0 GP1 => initial (snd GP0) /\ initial (snd GP1)
  end.

Fixpoint Program_to_State (P : Program) : State :=
  match P with
    | comm (bang' A) => (bar A) :: nil
    | comm (quer' A) => (hat A) :: nil
    | sequ P0 P1 => Program_to_State P0
    | par P0 P1 => Program_to_State P0 ++ Program_to_State P1
    | _ => nil
  end.

Definition exact (P : Program) (s : State) : Prop :=
  eq_State (Program_to_State P) s.
```

The $initial$ property is used to identify Programs that are not in the middle of any communication handshake. The $valid$ property is used to ensure that Programs may only be in the middle of a communication handshake at a current point of execution in the Program. Note that $initial\ P$ implies $valid\ P$, and that $valid\ P$ is an invariant of both semantics.

$exact\ P$ is also an invariant, specifically of the state semantics. Establishing this as an invariant is very important, as it expresses that one can infer the exact State of any $valid$ Program that once was $initial$. This ties in precisely with the overall ability to perform our program analysis using a trace semantics that is not concerned with state.

ProofStateToTraceMap.v

This file provides a mapping from steps to traces through the following definitions:

```
SS_to_Label (ss : SmallStep P s P' s') : Label := …

SS_to_Trace (ss : SmallStep P s P' s')
 : Trace P (SS_to_Label ss) P' := …

ES_to_Label (es : ErrorStep P s) : Label := …

ES_to_Program (es : ErrorStep P s) : Program := …

ES_to_Trace (es : ErrorStep P s)
 : Trace P (ES_to_Label es) (ES_to_Program es) := …
```

Defined here is just a simple structural mapping from the state semantics to the *open-world* semantics, whereas the files to follow must exhibit more complex reasoning about the classification functions of the closed-world semantics (compliance, interference and stability) with respect to this mapping.

ProofStateToTraceBase.v

This file provides definitions and proofs common to `ProofStateToTraceCompliance.v`, `ProofStateToTraceInterferance.v`, and `ProofStateToTraceStability.v`. Some notable constructs are:

```
Fixpoint es_interferant {P s} (es : ErrorStep P s) : Prop :=
  match es with
    | CommB2_E _ _ _ _ => True
    | CommQ2_E _ _ _ _ => True
    | Sequ_E _ _ _ es' => es_interferant es'
    | ParL_E _ _ _ es' => es_interferant es'
    | ParR_E _ _ _ es' => es_interferant es'
    | _ => False
  end.

Fixpoint es_nondetreq {P s} (es : ErrorStep P s) : Prop :=
  match es with
    | Sequ_E _ _ _ es' => es_nondetreq es'
    | ParL_E _ _ _ es' => es_nondetreq es'
    | ParR_E _ _ _ es' => es_nondetreq es'
    | Det2_E _ _ _ _ _ => True
    | _ => False
  end.

Fixpoint es_nonstable {P s} (es : ErrorStep P s) : Prop :=
  match es with
    | Sequ_E _ _ _ es' => es_nonstable es'
    | ParL_E _ _ _ es' => es_nonstable es'
    | ParR_E _ _ _ es' => es_nonstable es'
    | ParL_NSE _ _ _ _ _ _ _ => True
    | ParR_NSE _ _ _ _ _ _ _ => True
    | _ => False
  end.
```

which indicate the type of trace error exhibited by an ErrorStep.

ProofStateToTraceCompliance.v

Through the following definitions, this file proves that the previously defined mapping from steps to traces generates compliant traces:

```
SS_to_Compliant (ss : SmallStep P s P' s') (e : exact P s)
 : compliant (SS_to_Label ss) := …

ES_to_Compliant (es : ErrorStep P s) (e : exact P s)
 : compliant (ES_to_Label es) := …
```

Note that both non-erroneous and erroneous steps map to compliant traces, consistent with the closed-world trace semantics. The key insight here is that the starting State of any SmallStep is equal to the Probe list $pre\ a$, where a is the Label of the Trace to which that SmallStep is mapped.

ProofStateToTraceInterferance.v

Through the following definitions, this file proves that the previously defined mapping maps non-erroneous steps to non-interferant traces and interferant steps to interferant traces:

```
SS_to_notInt (ss : SmallStep P s P' s') (e : exact P s)
 : interferant (SS_to_Label ss) -> False := …

ES_to_Int (es : ErrorStep P s) (e : exact P s) (i : es_interferant es)
 : interferant (ES_to_Label es) := …
```

One step in completing the first of these two proofs was recognizing the following invariant across non-erroneous SmallSteps: that if a Probe appears in $down\ a$ where a is the Label of the Trace to which the SmallStep is mapped, then that Probe appears in $down\ a$ exactly once. This invariant proved to be helpful in defining SS_to_notInt.

ProofStateToTraceStability.v

Through the following definition, this file proves that the previously defined mapping maps non-stable steps to non-stable traces:

```
ES_to_notStable (es : ErrorStep P s) (ns : es_nonstable es)
                (v : valid P) (e : exact P s)
 : ~ stable (ES_to_Label es) := …
```

Dual to that of proving compliance, the key insight here is that the ending State of any SmallStep is equal to the Probe list $post\ a$, where a is the Label of the Trace to which that SmallStep is mapped.

Note that the closed-world trace semantics appropriately do not require us to prove that non-erroneous steps map to stable traces.

ProofStateToTrace.v

In correspondence with the first theorem of **Section 3.3**, this file maps non-erroneous step sequences to non-erroneous trace sequences and erroneous steps to erroneous traces as follows:

```
BigStep_to_BigTrace (bs : BigStep P s P' s') (v : valid P) (e : exact P s)
 : BigTrace P P' := …

ErrorStep_to_ErrorTrace (es : ErrorStep P s) (v : valid P) (e : exact P s)
 : ErrorTrace P := …
```

The state-to-trace direction of the semantics equivalence proof was by far the easier of the two, generally because of the fact that the state semantics are less expressive than the trace semantics in that they cannot capture concurrent executions of different parts of a parallel program. We'll see in the opposite direction that the final trace-to-state mapping is not as straightforward as the types above.

ProofTraceToStateBase.v

This file provides definitions and proofs common to `ProofTraceToStateInterferance.v`, `ProofTraceToStateStability.v`, and `ProofTraceToStateMain.v`. We refer the reader to the full Coq code for more details.

ProofTraceToStateInterferance.v

Through the following definition, this file maps interferant traces to erroneous steps:

```
TraceInt_to_ES (t : Trace P a P') (c : compliant a) (i : interferant a)
 : ErrorStep P (pre a) := …
```

Mapping interferant traces to erroneous steps proved to be the simplest and most straightforward piece of the trace-to-state direction of the proof.

ProofTraceToStateStability.v

Through the following definition, this file maps non-stable traces to erroneous steps:

```
TraceNotStable_to_ES (t : Trace P a P') (c : compliant a) (ns : ~ stable a)
 : ErrorStep P (pre a) := …
```

Mapping non-stable traces to erroneous steps proved to be a good amount more complicated than that of interferant traces. This is partly because of the way non-stable erroneous steps are defined: through the nested induction of Nonstable in ErrorStep. The insights that lead to the final construction involved identifying several different complex invariant relationships between the different functions on Labels (e.g. pre, $post$, up, $down$, etc.).

ProofTraceToStateMain.v

Through the following definition, this file maps compliant traces to non-erroneous or erroneous step sequences:

```
Trace_to_BSorBSES (t : Trace P a P') (v : valid P) (c : compliant a)
 : sum (BigStep P (Program_to_State P) P' (Program_to_State P'))
       {P'' : Program & {s'' : State &
        prod (BigStep P (Program_to_State P) P'' s'')
             (ErrorStep P'' s'')}} := …
```

The trace semantics are more expressive than the state semantics in that they can capture multiple concurrent Program execution steps in a single trace. An immediate implication of this difference is that in general, a single trace must map to a state step sequence (rather than a single step) in which any concurrent execution steps are broken up into single sequential steps. Another less obvious implication is that given a non-erroneous trace, it may map to a non-erroneous or erroneous state step sequence. This is because certain non-erroneous concurrent executions do not have non-erroneous sequential orderings of those executions.

This difference in expressiveness between the two semantics results in the Coq construct above. While we may opt to rephrase our theorem as a stronger relationship between strictly non-erroneous traces and state step sequences, the definition above is suitable for our needs, which are geared toward that between *erroneous* traces and state step sequences. The logical dependency that this file has on `ProofTraceToStateStability.v` exists because the general compliant traces found to map to erroneous step sequences do so in the same fashion as that of traces known to be non-stable (which are handled in said file). These noteworthy aspects of the proof are accented with dark red in the Visual Overview.

ProofTraceToState.v

In correspondence with the second theorem of **Section 3.3**, this file maps compliant trace sequences to non-erroneous or erroneous step sequences and erroneous traces to erroneous steps as follows:

```
BigTrace_to_BigStep (v : valid P) (bt : BigTrace P P')
 : sum (BigStep P (Program_to_State P) P' (Program_to_State P'))
       {P'' : Program & {s'' : State &
        prod (BigStep P (Program_to_State P) P'' s'')
             (ErrorStep P'' s'')}} := …

ErrorTrace_to_ErrorStep (et : ErrorTrace P)
 : ErrorStep P (Program_to_State P) := …
```

While the trace-to-state direction of the semantics equivalence proof results in types less elegant than that of the state-to-trace direction, the constructs shown here are nevertheless sufficient to prove our theorem.

ProofFinal.v

In correspondence with the corollary in **Section 3.3**, this file proves that erroneous step and trace sequences map to each other as follows:

```
BigErrorStep_iff_BigErrorTrace (i : initial P)
 : BigErrorStep P <-> BigErrorTrace P := …
```

The final construct shows here confirms that for any $initial$ Program P, there exists an erroneous sequence in the state semantics (starting from the initial state) if and only if there exists an erroneous sequences in the trace semantics.

In this file, we additionally include the line “Print Assumptions BigErrorStep_iff_BigErrorTrace” which upon execution prints the only assumptions made in building the construct above, which are the assumptions of a Channel type and decidable equality on that type.