A Lazy Approach to Compositional Verification*

N. Shankar
Computer Science Laboratory
SRI International
Menlo Park CA 94025 USA
Phone: (415)859-5272 `shankar@csl.sri.com`

Technical Report CSL-93-08, December 1993

Abstract

The verification of concurrent systems with a large number of interacting components poses a significant challenge to existing verification methodologies. We present a simple approach to compositional verification that allows global system properties to be established by considering only the behavior of the relevant system components. Each component specification describes the transitions due to the component itself and constrains the transitions due its environment. The composition of two specifications is an *interleaving conjunction* of the two specifications. Every property derived from a component specification remains true when this component is composed with other components. This compositional approach to verification is illustrated with the example of a buffer composed from two smaller buffers, and a real-time controller for a railroad crossing system composed from the subsystems for trains, the controller, and the gate. These examples have been verified using the PVS verification system.

1 Introduction

We consider the question of how one might model and prove properties of a large distributed system such as a railroad network consisting of a large number of tracks, trains, and railroad crossings. One fairly conventional way to model such a system is with a global state indicating the status of the tracks and crossings, and the positions of the trains. Transitions represent the changes to the system state due to a change in the status of a track or a crossing, or a change in the position of a train. The desired properties of such a system can be captured by means of assertions on the states that are reachable by means of the transitions. A common form of a system property is an invariance property stating that a given assertion is true in every reachable state of the system. Invariance properties are

*This work was supported by the US Naval Research Laboratory under contract N00014-92-C-2177

typically established by showing that the given assertion is initially true and preserved by *each* of the transitions. When such an invariant is affected only by a few components of the overall system, it is clearly wasteful to verify the invariance with respect to *all* of the transitions of the system. For example, some properties of the railroad network merely concern the subsystem consisting only of the trains, while other properties involve the interaction between the trains and the controller. The verification of such complex interacting systems will be unmanageable unless there are techniques for ensuring that proofs of system properties depend only on the transitions that are relevant to the property. This report presents a compositional verification technique for managing the proofs of concurrent and real-time systems. This technique has been mechanized using the PVS verification system [OSR93] and has been applied to some small examples.

As an illustration of the need for compositionality, one useful but simple invariance property [Sha93] of a safe railroad network is that the controller signals for the gate to be lowered within a bounded delay from when a train is first detected as approaching the crossing. Only the train and the controller are relevant to the proof of this invariant since the gate has no effect on the train or the controller signal. In several previous approaches, including our own, this invariant had to be proved relative to every transition of the entire system. The problem is that there could be a large number of transitions, many of which are irrelevant to the invariant in question. There is yet another problem with the lack of compositionality: if each invariance property must be preserved by every transition, then each transition must be specified in sufficient detail to validate every such invariant. That is, they must indicate how each value in the global state is changed or not changed by the transition. This conflicts with the need to avoid excessive specificity in formalizing system transitions. If we can establish global system properties by verifying them locally against the relevant system components, we could both simplify the specification as well as proofs about the specification.

Our approach to the verification of concurrent, real-time systems begins by considering a behavior as an execution trace of the system. A behavior is specified by a program in terms of the valid initial states of the system and the valid transitions from one system state to another. A program thus identifies a set of behaviors. A program property is also given by a set of behaviors. A program satisfies a property if the set of behaviors corresponding to the program is a subset of those behaviors satisfying the given property. One way of composing two such programs is take their *conjunction* so that we get the intersection of the two sets of behaviors. This has the nice property that any property of a component program in a composite program is also a property of the composite program. The problem with composition as conjunction is that each transition of the composite program must simultaneously be a transition of its components. When transitions are combined in this manner, they can often interfere with each other in unintended ways. Yet another problem with conjoining programs is that the number of nondeterministic actions in the resulting program can be the product of the numbers of possible actions in each individual component. To avoid these problems programs are specified as sets of behaviors where each behavior included transitions due to the environment. Each program can place constraints on the environment transitions. The composition of two programs is defined to ensure that the transition of one program is an environment transition of the other. This form of composition still preserves the properties of each component without compounding the nondeterminism or introducing unintended interactions between the components.

In our approach, each program or program component is specified in three parts consisting of the initialization predicate I, an atomic action G, and another atomic action R. The initialization predicate I constrains the initial state of any trace satisfying the program. The atomic actions G and R represent transitions that are binary relations between states. Each trace satisfying the given program must be such that every pair of adjacent states in the trace satisfies either G or R. The G action is the atomic action *guaranteed* by the component, and the R action is what the component *relies* on from the atomic actions of its environment. Note that the environment action R is essentially a safety constraint on the environment and asserts that any atomic state change due to the environment must respects the constraint specified in R. The action R is often referred to as a rely condition or constraint.

The composition $P_1 \| P_2$ of two components, P_1 given by $\langle I_1, G_1, R_1 \rangle$ and P_2 given by $\langle I_2, G_2, R_2 \rangle$, is the program $\langle I_1 \wedge I_2, (G_1 \wedge R_2) \vee (G_2 \wedge R_1), R_1 \wedge R_2 \rangle$. The conjunction of predicates is the pointwise conjunction of the predicates applied to a given state, and the conjunction of actions is similarly the pointwise conjunction of the actions applied to a given pair of states. The intuition behind the definition of composition is that the composed system is an interleaving of the G_1 action with the R_2 constraint and the G_2 action with the R_1 constraint. When G_1 and G_2 are specified in sufficient detail that G_1 implies R_2 and G_2 implies R_1, we can simplify the composition suitably. However, we should not always expect such a simplification since G_1 as specified may have no bearing on R_2. Typical rely conditions will assert that the values of certain variables are held fixed, or that certain values can only be changed in a certain way by the environment. It is possible, for example, for the conjunction of G_1 and R_2 to be inconsistent, in which case component 1 violates the constraints specified by component 2. We can of course derive any property from such an inconsistent system specification, but there is no way to refine such a specification down to a sound implementation.

We have used PVS [OSR93] to prove that the above notion of composition is associative and commutative. This composition operation is not necessarily idempotent. We have also proved the crucial fact that every property of a component is preserved when it is composed with other components. This fact makes it possible to prove a global property with respect to some relevant subset of the system components.

The specification of a component in terms of the interleaving of component and environment actions is not novel. This approach has been employed in several previous compositional methodologies. What is novel in our approach is that the composition of one component with another is obtained by merely conjoining the guarantee action of one with the rely constraint of the other. This form of composition does not generate any proof obligations in contrast to the rely-guarantee approaches where composition generates the proof obligation that the guarantee action of one component must entail the rely constraint of the other. Our primary objection to the generation of such proof obligations is that they can only be discharged by specifying the guarantee actions in excessive detail. Our approach argues for a separation of concerns where composition is simple and useful operation with the expectation that the specification will eventually be checked for consistency perhaps by refining it to down an implementation.

Related Work. The earliest approaches to composition operated by demonstrating *noninterference* to show that each component preserves the truth of any assertions within another component. This approach is combinatorially explosive since it essentially amounts to proving certain invariants in a global manner. It is exactly this combinatorial blowup that our approach seeks to avoid. Other related work by Abadi–Lamport [AL90] and by Collette [Col93] considers composition of specifications by means of conjunction: the challenge here is to try to find conditions under which specifications can be composed. For example, Collette adapts the Abadi–Lamport approach to the Unity logic [CM88]. Here a composition $P_1 \| P_2$ can guarantee a property G provided the environment ensures R, when

1. Each component P_i guarantees G_i provided R_i, and
2. The behaviors satisfying R together with the safety part G_j^S of G_j also satisfy R_i, for $i \neq j$. In other words, the environment for the composition together with the safety part of the other component j must satisfy the environment assumption made by a component i. Such proof obligations are called *rely-guarantee* proof obligations.
3. The behaviors satisfying the component guarantees G_1 and G_2 together with the environment assumption R must satisfy the guarantee property G of the composition. Similarly, the behaviors satisfying G_1^S, G_2^S, and R also satisfy the safety property G^S.

The actual composition rule described in [Col93, AL90] is somewhat more complicated than the one described above. There is also a notion of an agent, to separate out those agents that are part of the environment and those that are in the component of interest.

In the work of Chang, Manna, and Pnueli [CMP94], a system specification is decomposed into modules with interfaces. The interface for a module specifies whether a variable is *external* and hence only changed by the environment and not the module, or *out* in which case it is only changed by the module and left unchanged by the environment. The composition of two modules is only permitted when their interfaces have been checked for compatibility by ensuring that an *out* declaration in one module is an *external* declaration in the other module. The advantage of the interface specifications is that it is easy to check them for compatibility so as to ensure that the composition of modules is consistent if each individual module specification is consistent. To verify the validity of a property with respect to a module, the module transitions are interleaved with those of the environment. This use of environment transitions is common to several approaches including our approach here.

We can now highlight some of the differences between our lazy approach to compositional verification with that of Abadi–Lamport–Collette (ALC) and Chang–Manna–Pnueli (CMP). In the ALC approach, the rely conditions are safety properties constraining the behavior of the environment. In our approach, we merely constrain the individual actions of the environment since the typical rely conditions can be stated in this form. We make no references to agents, whereas the ALC approach uses agents to distinguish the part of the behavior that is due to a given component from that part that is due to its environment. The CMP approach requires that each component annotate system variables according to whether they serve as inputs from the environment or outputs to the environment. We avoid such annotations and instead explicitly constrain the manner in which the environment of a component can change the value of a variable. In other words, the interface constraints are part of the same mechanism that is used to place more general rely constraints on the environment. The ALC proof rule is used to discharge rely-guarantee proof obligations when two components are composed: the guarantee properties of each component must imply the rely constraints of the other component. We do not generate or discharge any such proof obligations during composition. We simply treat the rely conditions on one component as additional constraints on the possible behaviors of another component in its environment. Note that we are dealing with components that are only partially specified and it is not always possible to discharge rely-guarantee proof obligations on the basis of such partial information. For example, component P_1 may rely on its environment to never decrease the value in variable x, where component P_2 neither mentions nor constrains the behavior of x. Abadi and Lamport [AL90] consider a noninterleaving form of composition using conjunction that makes composition combinatorially explosive. In our approach, the composition of two programs each with five atomic actions yields a program with 10 atomic actions, whereas the Abadi-Lamport notion of composition yields one with 25 atomic actions. We have also omitted consideration of liveness properties. These are proved in the customary manner in our approach : one cannot derive a liveness property by composing the liveness properties of each component since such an interdependence might be circular. In our approach, any property, including a liveness property, of a subsystem is also a property of the entire system.

Collette [Col93] proves the correctness of a buffer that is composed of two buffers. We describe a part of Collette's argument in Section 2. In Sections 3 and 4, we describe the corresponding proof using our approach. This proof has been verified using PVS.

2 A Simple Example of a Buffer

We examine a very simple example used by Collette [Col93] to make a few notions concrete. Collette's example is adapted from a similar example of Misra [Mis90]. Misra uses the bag data structure to give an abstract specification for a buffer. The buffer is then composed from two contiguous buffers communicating through a shared variable that is hidden from the environment. Misra also refines the bag data structure to a queue representation. Misra's proof composes the individual buffer specifications by explicitly demonstrating that neither buffer interferes with the safety properties of the other. Collette avoids the noninterference aspect of the proof by adapting the Abadi-Lamport composition principle to Unity. Collette's proof explicitly ensures that each component buffer respects the environment assumptions of the other.

In Collette's specification as shown in Figure 1, the buffer program P communicates with its environment through two variables in and out, and the state of the buffer resides in a variable b. The buffer relies on in and out being empty initially, and relies on its environment to insert values into in only when in is empty, and to remove values from out only when out is not empty. The variables in, out and b are otherwise left untouched by the environment. Collette adapts Unity for this purpose by subscripting the operators with the subset of agents to which an operator is applied. The notion of an agent need not be made precise here, but in the statements below, the buffer is composed of the agents μ, and its environment consists of the agents $\overline{\mu}$.

In the Unity notation, p `unless` q holds of the behavior of a program if whenever p holds in the behavior, p continues to hold until a point where q holds in the behavior. `stable` p can be defined using `unless` and it asserts that if p ever becomes true in the behavior, then it continues to remain true. `invariant` p asserts that p is initially true and stable. `constant` x asserts that x has a fixed value throughout the behavior. In Collette's notation, these operators are subscripted by a set of agents. Formally, the rely conditions R of the buffer are stated in a Unity style as

Figure 1: A Buffer and its Environment

1. **`initially`** $in = \perp$. The environment ensures that the input variable to the buffer is initially empty.

2. **`initially`** $out = \perp$. The environment similarly ensures that the output variable from the buffer is also empty.

3. **`initially`** $b = \perp$. The buffer itself is initially empty.

4. $out = m$ **`unless`**$_{\overline{\mu}}$ $out = \perp$. The environment does not change the value in *out* other than by reading it and setting *out* to $\perp$.

5. **`stable`**$_{\overline{\mu}}$ $out = \perp$. Once *out* has been read, the environment does not insert any values into *out*.

6. **`stable`**$_{\overline{\mu}}$ $in = m$. The environment only inserts values into *in* but does not otherwise change its value.

7. **`constant`**$_{\overline{\mu}}$ b. The environment does not affect the value of the buffer b.

The buffer guarantees that it will read in values from *in* only when *in* is not empty, and write out values to *out* only when *out* is empty. It also guarantees that no values are lost by ensuring that *in* appended to b appended to *out* is left unchanged by the buffer. A bit of reasoning is required to ensure that this latter part of the specification entails the intended behavior of the buffer, but this reasoning is overlooked by Collette. In particular, the correctness of the buffer is implied by the above condition only when we have the additional constraint that the buffer is unchanged by the environment. These guarantee conditions G are formally stated as

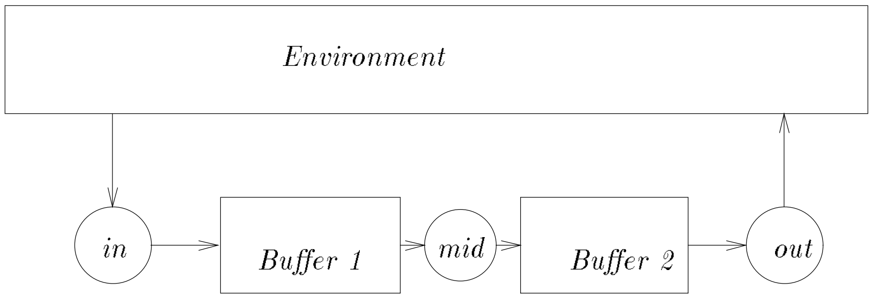

Figure 2: A Composite Buffer

1. $in = m$ $\texttt{unless}_\mu$ $in = \perp$
2. $\texttt{stable}_\mu$ $in = \perp$
3. $\texttt{stable}_\mu$ $out = m$
4. $\texttt{constant}_\mu$ $\langle in \rangle \bullet b \bullet \langle out \rangle$

The buffer as specified above can be obtained as the composition of two buffers P_1 and P_2 that have a similar specification as that for P above. This is shown in Figure 2. The agents corresponding to P_1 and P_2 are labeled μ_1 and μ_2, respectively. The variable *mid* replaces *out* in P_1 and *in* in P_2. The respective buffer values are maintained in the variables b_1 and b_2. The resulting rely conditions for buffer i are labeled R_i, and the corresponding guarantee conditions are labeled G_i. P_1 reads in values placed by the environment from *in*, and outputs values to *mid*. P_2 reads values from *mid* and outputs values to the environment via *out*. There is an additional rely constraint on the environment that the value of *mid* must not be affected by the environment:

1. $\texttt{initially}$ $mid = \perp$
2. $\texttt{constant}_{\overline{\mu}}$ mid

Collette now uses the composition principle to establish that the program resulting from the composition $P_1 \| P_2$ behaves like a buffer. The composition principle leads to four proof obligations requiring that

$$R, G_2^S \quad \vdash \quad R_1 \tag{1}$$

$$R, G_1^S \vdash R_2 \tag{2}$$

$$R, G_1^S, G_2^S \vdash G^S \tag{3}$$

$$R, G_1, G_2 \vdash G \tag{4}$$

Here G^S is the safety part of G, but since we have omitted the liveness aspects of the specification, G^S is just G. The proof rules of Unity can be employed to discharge each of the proof obligations. Since R_1 and R_2 each contain seven rely conditions, there are fourteen proof obligations resulting from 1 and 2, and about four other proof obligations from 3 and 4. Some of these proof obligations are not entirely trivial. In proving these obligations, the environment agents for the first buffer have to be partitioned into those of the overall system environment and the agents μ_2. We then have to demonstrate of the agents μ_2 that they only read from mid and do not otherwise change the values of in, b_1, or mid. There are similar proof obligations corresponding to P_2.

Collette also treats liveness properties that we have omitted from the present treatment. In addition, Collette deals with bounded buffers, but nothing essential is lost in restricting our attention to unbounded buffers.

3 A Lazy Approach to Composition

We now illustrate our approach to the compositional verification of the buffer. Our specification of this example is somewhat different from that of Misra and Collette. We specify the correctness of the buffer by asserting that the sequence of values output through *out* must be a prefix of the sequence of values input via *in*. We feel that this specification of the buffer captures the correctness property of the buffer more accurately than the one used by Collette.

As already mentioned, a program is specified as a triple consisting of an initialization predicate, a guarantee action, and a rely action. The buffer consists of the variables *inp* and its history variable *inph* represented as a list, the output *out* and the output variable *outh* represented as a list, and the buffer contents b which is also represented as a list. The $\bullet$ operation represents the concatenation of two lists. The operations fst and deq are such that for a nonempty list l, $deq(l) \bullet fst(l) = l$.

The initialization predicate for the buffer asserts that:

- b and *outh* are empty lists
- *out* contains the $\perp$ value indicating that it is initially empty

- if inp is $\perp$, then $inph$ is an empty list, otherwise $inph$ is a singleton list containing the initial value of inp

The guarantee action of the buffer is the pointwise disjunction of an input action and an output action. The input action enqueues a non-$\perp$ value from inp onto b and sets inp to $\perp$. The input action leaves out and $outh$ unchanged. The output action is enabled only when out is $\perp$ and b is nonempty. It dequeues a value from b and sets out to this value and simultaneously inserts this value into $outh$. The output action leaves inp and $inph$ unchanged.

The rely action of the buffer constrains any environment action to leave b and $outh$ unchanged. In addition, the environment can either insert a non-$\perp$ value into inp when inp is $\perp$ and $inph$ is simultaneously updated to contain the new inp value while out is unchanged, or the environment reads a non-$\perp$ value in out and sets out to $\perp$ while leaving inp and $inph$ unchanged, or the environment leaves inp, $inph$, and out also unchanged.

The specification can be given more formally as

1. $$\begin{array}{lll} I & \triangleq & b = outh = null \\ & \wedge & (inp = \perp \supset inph = null) \\ & \wedge & (inp \neq \perp \supset inph = \langle inp \rangle) \end{array}$$

2. $$\begin{array}{lll} READ & \triangleq & in \neq \perp \\ & \wedge & b' = \langle in \rangle \bullet b \\ & \wedge & inph' = inph \\ & \wedge & out' = out \\ & \wedge & outh' = outh \end{array}$$

3. $$\begin{array}{lll} WRITE & \triangleq & b \neq null \\ & \wedge & out = \perp \\ & \wedge & b' = deq(b) \\ & \wedge & out' = fst(b) \\ & \wedge & outh' = \langle fst(b) \rangle \bullet outh \\ & \wedge & inp' = inp \\ & \wedge & inph' = inph \end{array}$$

4. $G \triangleq READ \vee WRITE$

5. $$\begin{array}{lll} INPUT & \triangleq & inp = \perp \\ & \wedge & out' = out \\ & \wedge & (inp' = \perp \supset inph' = inph) \\ & \wedge & (inp' \neq \perp \supset inph' = \langle inp \rangle \bullet inph) \end{array}$$

6. $OUTPUT \triangleq out \neq \bot$
$\wedge\ out' = \bot$
$\wedge\ inp' = inp$
$\wedge\ inph' = inph$

7. $STUTTER \triangleq out' = out$
$\wedge\ inp' = inp$
$\wedge\ inph' = inph$

8. $R \triangleq b' = b$
$\wedge\ outh' = outh$
$\wedge\ (INPUT \vee OUTPUT \vee STUTTER)$

9. $P \triangleq \langle I, G, R \rangle$

Using PVS, we can prove using conventional techniques to prove that the above specification of the buffer maintains the invariant

$$inph = \langle inp \rangle \bullet reverse(b) \bullet outh. \tag{5}$$

This is the main safety property of the buffer.

Now we consider two instances of a buffer program above: P_1 with buffer variable b_1 and P_2 with buffer variable b_2. As shown in Figure 2, the two buffers communicate via the variable *mid* (and its history variable *midh*), which serves as the output of buffer 1 and the input to buffer 2.[1] For these two buffers we have the invariant 5 for buffer 1 with *out* and *outh* replaced by *mid* and *midh*, and similarly for buffer 2 with *in* and *inh* replaced by *mid* and *midh*. We can now easily prove the invariant 5 for the composed buffer from the invariants for the individual buffers by applying the associativity of the $\bullet$ operation:

$$\begin{aligned}
& inph \\
= \ & \{(5) \text{ for buffer } 1\} \\
& \langle inp \rangle \bullet reverse(b_1) \bullet midh \\
= \ & \{(5) \text{ for buffer } 2\} \\
& \langle inp \rangle \bullet reverse(b_1) \bullet \langle mid \rangle \bullet reverse(b_2) \bullet outh
\end{aligned}$$

[1]In an earlier version of the specification, following Collette's specification, we had placed an additional constraint on the environment to leave the value of *mid* unchanged, but this constraint was found to not be needed in the proof. The reason for this is that buffer 1 already requires that its environment can only remove values from *mid* and buffer 2 requires that its environment can only place values into *mid*, so that the combined specification then ensures that the combined environment cannot affect the value of *mid*.

$$
\begin{aligned}
= \quad & \{\textit{associativity}, \text{ and let } b = b_2 \bullet mid \bullet b_1\} \\
& \langle inp \rangle \bullet reverse(b) \bullet outh
\end{aligned}
$$

Note that the safety property of the composed buffer does not rely on any operational reasoning and is derived solely from the safety properties of the individual buffers. The safety property of each individual buffer is easily proved by operational means, that is, by showing that it holds initially and is preserved by each of the transitions of the buffer or its environment. Clearly we have not yet done the work of discharging the proof obligations that are explicit in Collette's proof, which is why our proof is so utterly simple. Indeed, these proof obligations cannot be discharged at this point in the specification since the component specifications do not contain enough information. These proof obligations need to be ensured during refinement. We feel that this separation of concerns is essential from the viewpoint of obtaining abstract specifications and straightforward proofs. If each component of a system is required to specify enough detail to discharge any rely conditions imposed by other components, the resulting specification could become overly specific and detailed. At the abstract level, it is reasonable to simply accumulate these environment constraints given by composition, and guarantee that these constraints are respected during the refinement process.

We also have no notion of agents in our methodology which means that we have no way of even stating the the main safety property of the buffer given by Misra and Collette. As observed earlier, their statement does not capture the main safety property of the buffer unless the environment is constrained to leave the value of the buffer variable unchanged.

Our approach does not preclude the possibility that there will be special cases where the guarantee conditions of one program are strong enough to discharge the rely constraints of another. In this case, the composed program can be simplified to eliminate those rely conditions that are easily discharged. The key point in our approach is that we do not require that any rely-guarantee proof obligations be discharged during composition. The advantage of avoiding these proof obligations is that the component specifications can be relatively weak, whereas stronger specifications will be needed to discharge rely-guarantee proof obligations. The price of the laziness during composition is that the composed specification could be inconsistent. The consistency of the specification is a proof obligation that will have to be established in the final stage of refinement. This separation of concerns between composition of specifications (which should be an easy and useful operation) and the consistency of the composed specification (which needs to be established in any case) makes this lazy form of composition easy to use.

4 PVS Development of the Buffer Proof

The safety property for the composite buffer is developed over five theories. The first theory, *state*, formalizes concurrent program behaviors in terms of subsets of traces. The second theory, *compose*, formalizes program composition as described above. The third theory is a simple formalization of the list operations *append* and *reverse* and some of their properties. The fourth theory builds on the previous ones to define a buffer program and prove its primary safety property. The final theory in the development proves the safety property of a buffer defined by composing two buffers.

4.1 Behaviors

The theory *state* introduces an uninterpreted type *state*. A program trace is a sequence of states that is a function mapping the type of natural numbers *nat* to *state*. A program action given by the type *Action* is a binary relation on states: it relates the state prior to the action to the state following the action. An assertion or *statepred* is a predicate on *state*. As defined by *Inv*, an assertion is an invariant on a trace if it holds on every state of the trace. *Init* defines an assertion to be initially true of a trace if it is true of the first state in the trace.

$state$: THEORY
BEGIN

$state$: TYPE

USING $sequences[state]$

$statepred$: TYPE $= PRED[state]$

$Action$: TYPE $= PRED[[state, state]]$

pp : VAR $statepred$

$action$: VAR $Action$

$seq, seq1, seq2$: VAR $sequence[state]$

aa, bb, cc : VAR $sequence[state]$

$Inv(pp)(aa) : bool = (\forall\ (n : nat) : pp(aa(n)))$

$Init(pp)(aa) : bool = pp(aa(0))$

END $state$

4.2 Composition

The theory *compose* builds on the theory *state* to define the key notion of composition. We describe the theory *compose* in two parts. The first part defines the notions of program and program behavior, and the second part defines the composition operation. An environment action *EAction* is a reflexive binary relation so as to permit the environment to possibly leave the state unchanged. A program is an element of the type *Program* and is a record consisting of an initialization assertion *init*, a guarantee action G, and an environment action R. A trace aa is a run of a program P_1, that is $aa \models P_1$ if the *init* predicate is true of the initial state of aa, and each adjacent pair of states in aa is related by the G or the R predicate. This means that each trace satisfying a program must start in an acceptable initial state and each transition in the trace must be that of the program or its environment.

```
compose : THEORY
  BEGIN

  USING state

  aa, bb, cc : VAR sequence[state]

  s, s_0, s_1 : VAR state

  EAction : TYPE  = {a : Action | (∀ s : a(s, s))}

  Program : TYPE  = [# init : statepred, G : Action, R : EAction#]

  P_1, P_2, P_3 : VAR Program

  aa |= P_1 : bool =
    init(P_1)(aa(0))
      ∧ (∀ (n : nat) : G(P_1)(aa(n), aa(n + 1)) ∨ R(P_1)(aa(n), aa(n + 1)))
```

The program resulting from the composition of two programs P_1 and P_2 is given by $P_1 \| P_2$. The resulting *init* predicate is the pointwise conjunction of the *init* predicates of P_1 and P_2. The G action of the composed program is essentially the pointwise disjunction of the individual G actions where each G action has been constrained by the R action of the other component. The R action of the composed program is pointwise conjunction of the R actions of the individual components. The rationale for this form of composition is

1. The environment constraint of the composed program combines the environment constraints of the components since the environment of the composed program is also part of the environment of each component.

2. An action of the composed program could be an action of either component. Since each component, say P_1, is part of the environment of the other component P_2 of the composition, the actions of P_2 are constrained by the environment constraints of P_1. We cannot say that the actions of P_2 meet the environment constraints imposed by P_1 since, for example, P_1 and P_2 might have no observable state variables in common.

The most important property of *Compose* is given by the lemma *deCompose1*, which asserts that any trace satisfying $P_1 \| P_2$ already satisfies P_1 (and P_2, by commutativity). This property is quite easily proved from the definitions of *Compose* and *Run*. It is important because it implies that *every* property of a component P_1 is preserved in the composition $P_1 \| P_2$. A *property* Q of a program P_1 is a predicate on traces such that $aa \models P_1$ implies `Q(aa)`. The *Compose* operation is also easily proved to be commutative and associative.

$$
\begin{array}{l}
(P_1 \| P_2) : \ \mathit{Program} \ = \\
\quad (\# \ \mathit{init} := \ (\lambda \ s : \ \mathit{init}(P_1)(s) \ \wedge \ \mathit{init}(P_2)(s)), \\
\qquad G := \\
\qquad\ (\lambda \ s_0, \\
\qquad\qquad s_1 : \ (G(P_1)(s_0, s_1) \ \wedge \ R(P_2)(s_0, s_1)) \\
\qquad\qquad\quad \vee \ (G(P_2)(s_0, s_1) \ \wedge \ R(P_1)(s_0, s_1))), \\
\qquad R := \ (\lambda \ s_0, s_1 : \ R(P_1)(s_0, s_1) \ \wedge \ R(P_2)(s_0, s_1)) \ \#)
\end{array}
$$

$\mathit{deCompose1}$: LEMMA $aa \models (P_1 \| P_2) \ \supset \ aa \models P_1$

$\mathit{Compose\_commutes}$: LEMMA $(P_1 \| P_2) \ = \ (P_2 \| P_1)$

$\mathit{Compose\_assoc}$: LEMMA
$((P_1 \| P_2) \| P_3) \ = \ (P_1 \| (P_2 \| P_3))$

END *compose*

4.3 Buffers

The theory *append* below defines the list operations *append* and *reverse*. These are used to define the buffer operations. Both *append* and *reverse* have fairly standard recursive definitions. The lemma *assoc* asserts the associativity of append. It is proved without manual interaction by the PVS proof strategy `induct-and-rewrite`, as is the lemma *append_null*. When the `induct-and-rewrite` strategy is applied to the lemma *reverse_append*, the proof is completed with two further interactive steps.

$append[T: \text{TYPE}]: \text{THEORY}$
  BEGIN

  $d, d_1, d_2:$ VAR $list[T]$

  $d_1 \bullet d_2:$ RECURSIVE $listT$
    $=$ IF $cons?(d_1)$ THEN $cons(car(d_1), cdr(d_1) \bullet d_2)$ ELSE d_2 ENDIF
   MEASURE $\lambda\ d_1 d_2: \ length(d_1)$

  $assoc:$ LEMMA $d \bullet d_1 \bullet d_2 \ = \ d \bullet d_1 \bullet d_2$

  $append\_null:$ LEMMA $d \bullet null \ = \ d$

  $reverse(d):$ RECURSIVE $list[T]$
    $=$ (IF $cons?(d)$ THEN $reverse(cdr(d)) \bullet cons(car(d), null)$
      ELSE $null[T]$
      ENDIF)
   MEASURE $length[T]$

  $reverse\_append:$ LEMMA $reverse(d_1 \bullet d_2) \ = \ reverse(d_2) \bullet reverse(d_1)$

  END $append$

The theory *buffer* below defines the buffer program and proves its primary safety property. This theory is described in five parts. The first part declares the needed types and program variables, the second part defines a few useful queue operations, the third part describes the buffer actions, the fourth part specifies the complete buffer program, and the main safety property of the buffer is proved in the final part. This theory takes as parameters the type T of the buffer elements, and a special value *empty* to indicate a null value. The predicates *empty_T?* and *nempty_T?*, respectively, indicate when a given value is or is not equal to *empty*. Program variables of type t are functions from the state type *state* to the type t. The PVS variables *inp* and *out* range over program variables of type T. The PVS variables b, *inph*, and *outh* range over program variables of type *BUF* (which is defined as $list[T]$) that will be treated below as history variables for *inp* and *out*, respectively.

$buffer[T:$ TYPE, $empty: T]:$ THEORY
BEGIN

USING $append, compose$

$s, s_0, s_1, s_2:$ VAR $state$

$x, y, z:$ VAR T

$empty\_T?(x): bool = (x = empty)$

$nempty\_T?(x): bool = \neg\ empty\_T?(x)$

$xx, yy, zz:$ VAR $(nempty\_T?)$

$BUF:$ TYPE $= list[T]$

$B:$ VAR BUF

$inp, out:$ VAR $[state \rightarrow T]$

$inph, outh:$ VAR $[state \rightarrow BUF]$

$b:$ VAR $[state \rightarrow BUF]$

The queue operations *enq*, *deq*, and *fst* are defined in terms of the list operations *append*, *cdr*, and *car*. The *enq* operation places an element at the end of a given list (used as a queue). The *deq* operation returns the result of removing the head of a given queue, whereas the *fst* operation returns the head of the given queue.

$$empty?(B) : bool = (null?(B))$$

$$nempty?(B) : bool = cons?(B)$$

$$enq(xx, B) : (nempty?) = B \bullet cons(xx, null[T])$$

$$deq((B : (nempty?))) : BUF = cdr(B)$$

$$fst((B : (nempty?))) : T = car(B)$$

The buffer program is given by defining its initialization predicate, its G action, and its R action. The initialization for the buffer is given by the *initbuf* operation, which ensures that

1. The buffer b is an empty queue (list).
2. The output variable *out* contains the null value *empty*, and the output history *outh* is empty.
3. Either the input variable *inp* contains a null value, in which case the input history *inph* is empty, or the input history is a singleton list containing the non-null value in *inp*.

The G action of the buffer program is the pointwise disjunction of an input action *inp_action* and an output action *out_action*. When *inp* contains a non-null value, the input action enqueues this non-null value at the tail of the buffer b and sets *inp* to be *empty*. This action leaves the variables *inph*, *out*, and *outh* unchanged.

When the buffer b is nonempty and *out* contains the null value, then the output action *out_action* removes the head of the buffer b and places it in *out*, updates the output history *outh* to contain this new value, and leaves *inp* and *inph* unchanged.

$$
\begin{array}{l}
\mathit{inp\_action}(\mathit{inp}, \mathit{inph}, \mathit{out}, \mathit{outh}, b)(s_1, s_2) : \ \mathit{bool} \ = \\
\quad \mathit{inp}(s_1) \ \neq \ \mathit{empty} \\
\qquad \wedge \ \mathit{inp}(s_2) \ = \ \mathit{empty} \\
\qquad\ \ \wedge \ \mathit{inph}(s_2) \ = \ \mathit{inph}(s_1) \\
\qquad\quad \wedge \ b(s_2) \ = \ \mathit{enq}(\mathit{inp}(s_1), b(s_1)) \ \wedge \ \mathit{out}(s_2) \ = \ \mathit{out}(s_1) \ \wedge \ \mathit{outh}(s_2) \ = \ \mathit{outh}(s_1)
\end{array}
$$

$$
\begin{array}{l}
\mathit{out\_action}(\mathit{inp}, \mathit{inph}, \mathit{out}, \mathit{outh}, b)(s_1, s_2) : \ \mathit{bool} \ = \\
\quad \mathit{nempty?}(b(s_1)) \\
\qquad \wedge \ \mathit{out}(s_1) \ = \ \mathit{empty} \\
\qquad\ \ \wedge \ \mathit{out}(s_2) \ = \ \mathit{fst}(b(s_1)) \\
\qquad\quad \wedge \ \mathit{outh}(s_2) \ = \ \mathit{cons}[T](\mathit{fst}(b(s_1)), \mathit{outh}(s_1)) \\
\qquad\qquad \wedge \ b(s_2) \ = \ \mathit{deq}(b(s_1)) \ \wedge \ \mathit{inp}(s_2) \ = \ \mathit{inp}(s_1) \ \wedge \ \mathit{inph}(s_2) \ = \ \mathit{inph}(s_1)
\end{array}
$$

$$
\begin{array}{l}
\mathit{initbuf}(\mathit{inp}, \mathit{inph}, \mathit{out}, \mathit{outh}, b)(s) : \ \mathit{bool} \ = \\
\quad (\mathit{empty?}(b(s)) \\
\qquad \wedge \ \mathit{empty\_T?}(\mathit{out}(s)) \\
\qquad\ \ \wedge \ \mathit{empty?}(\mathit{outh}(s)) \\
\qquad\quad \wedge \\
\qquad\quad (\text{IF } \mathit{empty\_T?}(\mathit{inp}(s)) \text{ THEN } \mathit{empty?}(\mathit{inph}(s)) \\
\qquad\qquad \text{ELSE } (\mathit{inph}(s) \ = \ \mathit{cons}(\mathit{inp}(s), \mathit{null})) \\
\qquad\qquad \text{ENDIF}))
\end{array}
$$

The R part given by *relybuf* of the buffer program is quite crucial to the correct operation of the buffer. The first part of the definition of *relybuf* asserts that any environment action must leave the buffer b and the output history *outh* unchanged. The second part of the definition of *relybuf* allows three possible actions by the environment while also respecting the constraints stated in the first part. The first action is when *inp* is empty; then, the environment can either leave *inp*, *inph*, and *out* unchanged, or can place a non-null value in *inp* and simultaneously into *inph* while leaving *out* unchanged. The second action allows the environment to remove a non-null value from *out* while leaving *inp* and *inph* unchanged. The third action allows the environment to leave all the observable variables of the buffer program unchanged. This last possibility is already implicit in the first action but is included for explicitly allowing the possibility that the environment is permitted to "do nothing." The entire buffer program is given by the definition of P.

$$
\begin{array}{l}
relybuf(inp, inph, out, outh, b)(s_1, s_2) : \ bool \ = \\
\quad b(s_2) \ = \ b(s_1) \\
\qquad \wedge \ outh(s_2) \ = \ outh(s_1) \\
\qquad \ \wedge \\
\qquad ((empty\_T?(inp(s_1)) \\
\qquad\qquad \wedge \ out(s_2) \ = \ out(s_1) \\
\qquad\qquad \ \wedge \\
\qquad\qquad (\text{IF } empty\_T?(inp(s_2)) \text{ THEN } inph(s_2) \ = \ inph(s_1) \\
\qquad\qquad\quad \text{ELSE } inph(s_2) \ = \ cons(inp(s_2), inph(s_1)) \\
\qquad\qquad\quad \text{ENDIF})) \\
\qquad\quad \vee \\
\qquad\quad (nempty\_T?(out(s_1)) \\
\qquad\qquad\quad \wedge \ out(s_2) \ = \ empty \ \wedge \ inp(s_2) \ = \ inp(s_1) \ \wedge \ inph(s_2) \ = \ inph(s_1)) \\
\qquad\qquad \vee \ (out(s_2) \ = \ out(s_1) \ \wedge \ inp(s_2) \ = \ inp(s_1) \ \wedge \ inph(s_2) \ = \ inph(s_1))) \\
\\
P(inp, inph, out, outh, b) : \ Program \ = \\
\quad (\# \ init := \ initbuf(inp, inph, out, outh, b), \\
\qquad G := \\
\qquad\quad (\lambda \ s_1, s_2 : \\
\qquad\qquad inp\_action(inp, inph, out, outh, b)(s_1, s_2) \\
\qquad\qquad\quad \vee \ out\_action(inp, inph, out, outh, b)(s_1, s_2)), \\
\qquad R := \ relybuf(inp, inph, out, outh, b) \ \#)
\end{array}
$$

The last part of the *buffer* theory is the statement and proof of the safety property of the buffer. This property as stated in the lemma *main* essentially asserts that the output history is always an initial segment of the input history. More specifically, it states that the input history can be obtained from the output history by appending any non-null input value in *inp* along with the contents buffer (in reverse order) to the output history. The expression *econs(x, b)* behaves like *cons(x, b)* when x is non-null, and returns b otherwise.

The proof of *main* is a routine exercise in inductively proving program invariance.

aa, bb, cc : VAR $sequence[state]$

$econs(x, (b : BUF)) : BUF \; = \; (\text{IF } empty\_T?(x) \text{ THEN } b \text{ ELSE } cons(x, b) \text{ ENDIF})$

$main$: LEMMA
$aa \models P(inp, inph, out, outh, b)$
$\supset Inv(\lambda\ s : inph(s) \; = \; econs(inp(s), reverse(b(s)) \bullet outh(s)))(aa)$

END $buffer$

4.4 Composing Buffers

In building a single buffer by composing two smaller buffers, we introduce some specific program variables. The individual buffers themselves are given by the program variables b_1 and b_2 of type BUF. The program variable *inp* serves as the input to buffer 1, *out* serves as the output to buffer 2, and *mid* serves as both the output of buffer 1 and input to buffer 2. Associated with *inp*, *out*, and *mid* are their respective history variables *inph*, *outh*, and *midh*.

The program for the buffer composed of the two individual buffers is given by

$$P(inp, inph, mid, midh, b_1) || P(mid, midh, out, outh, b_2)$$

We can now state and prove the analogue of the safety property given in *main* for the composed buffer. This argument follows the one shown in Section 3.

```
composed_buf[T : TYPE, empty : T] : THEORY
  BEGIN

  USING buffer[T, empty]

  s, s0, s1, s2 : VAR state

  x, y, z : VAR T

  xx, yy, zz : VAR (empty_T?)

  inp, out, mid : [state -> T]

  inph, outh, midh : [state -> BUF]

  b1, b2 : [state -> BUF]

  aa, bb, cc : VAR sequence[state]

  main_comp : LEMMA
    aa |= (P(inp, inph, mid, midh, b1)||P(mid, midh, out, outh, b2))
      ⊃
     Inv(λ s : inph(s)
                 =
               econs(inp(s),
                     reverse(b1(s)) • econs(mid(s), reverse(b2(s)) • outh(s))))(aa)

  END composed_buf
```

5 An Advanced Example: A Railroad Crossing Controller

This example has already been considered in detail in a previous technical report [Sha92]. Heitmeyer and Jeffords [HJ93] present a thorough discussion of this example with extensive references to the related literature. The verification approach used in that report was not compositional. As a consequence, the invariance proofs were carried out relative to all of the transitions in the system. The system transitions included those of the trains, the controller, and the gate. As already mentioned, for several of these properties, only one or two of the subsystems were relevant so that these proofs were needlessly complicated. We will briefly describe the specification of the railroad crossing system and then sketch the details of the compositional development of its safety property.

5.1 State and Time

The theory *time* below is an extension of the theory *state* from the previous example. It introduces the type *time* consisting of the nonnegative rational numbers. Then *Time* is declared as a program variable of type *time*. A trace is an element of the type *behavior* if it is a trace where the *Time* parameter is nondecreasing and non-Zeno. These latter concepts are expressed by the predicates *nondec?* and *nonzeno?*, respectively. The *compose* theory from the previous example is also employed in the current example.

$time:$ THEORY
BEGIN

$state:$ TYPE

USING $sequences[state]$

$statepred:$ TYPE $= PRED[state]$

$Action:$ TYPE $= PRED[[state, state]]$

$pp:$ VAR $statepred$

$action:$ VAR $Action$

$time:$ TYPE $= \{x: rational \mid x \geq 0\}$ CONTAINING 0

$Time: [state \rightarrow time]$

$seq, seq1, seq2:$ VAR $sequence[state]$

$i, j, k:$ VAR nat

$nondec?(seq): bool = (\forall\ i, j: i > j \supset Time(seq(j)) \leq Time(seq(i)))$

$nonzeno?(seq): bool = (\forall\ (x: rational): (\exists\ i: x < Time(seq(i))))$

$behavior?(seq): bool = nondec?(seq) \wedge nonzeno?(seq)$

$behavior:$ TYPE $= (behavior?)$

$aa, bb, cc:$ VAR $behavior$

$Inv(pp)(aa): bool = (\forall\ (n: nat): pp(aa(n)))$

$Init(pp)(aa): bool = pp(aa(0))$

END $time$

5.2 The Since Operator

Given an assertion pp and a trace aa, the expression $|pp|(aa(n))$ is the time difference between the *aa(m)* and *aa(n)* where m is the largest number below n such that pp(aa(m)) holds — in other words, $since(pp)(aa(n))$ returns the time that has elapsed since pp held in a state prior to $aa(n)$ in the trace aa. In the specification of the railroad crossing, the *since* operator is used, for example, to bound the time between when a train is detected as approaching, and the controller signals for the gate to be lowered.

The specification of the since operator given by $|pp|$ is a transcription of two Hoare axioms:

$$\{pp \wedge x = Time\}\ S\ \{|pp| = Time - x\}$$
$$\{\neg pp \wedge x = Time \wedge y = |pp|\}\ S\ \{|pp| = y + Time - x\}$$

The above axioms do not yield any information on the value of $|pp|$ in the initial state. The behaviors in the type *behavior0* are constrained to be such that the initial value of $|pp|$ is one more than the initial value of $Time$. This initialization essentially indicates that there is no state prior to the initial state where a predicate pp holds.

$since$: THEORY
  BEGIN

  USING $time$

  i, j, k : VAR nat

  pp : VAR $statepred$

  $action$: VAR $Action$

  aa, bb, cc : VAR $behavior$

  $|\star|$: $[statepred \rightarrow [state \rightarrow time]]$

  $since\_ax$: AXIOM
   $|pp|(aa(i + 1))$
     $=$
    (IF $pp(aa(i))$ THEN $(Time(aa(i + 1)) - Time(aa(i)))$
     ELSE $|pp|(aa(i)) + (Time(aa(i + 1)) - Time(aa(i)))$
     ENDIF)

  $behavior0$: TYPE $= \{aa : behavior \mid (\forall\, pp : |pp|(aa(0)) = Time(aa(0)) + 1)\}$

  END $since$

A number of interesting theorems can be proved of the *since* operator. These are enumerated in the theory *since_thms* which is presented below in two parts. These theorems have already been explained in an earlier report [Sha92] so the explanation will not be repeated here. The theorem *Inv_implies* asserts that an invariant implication yields an implication of invariants. The theorem *since_since* provides a way to accumulate successive upper bounds. The theorem *since_implies* asserts that the weaker assertion is true at least as recently as a stronger one. The definitions of the *min* and *max* operators are also given.

$since\_thms$: THEORY
BEGIN

USING $time, since$

i, j, k : VAR nat

pp, qq, rr : VAR $statepred$

aa, bb, cc : VAR $behavior0$

$rt, rt1, rt2, rt3$: VAR $time$

s, s_0, s_1 : VAR $state$

$Inv\_implies$: LEMMA $Inv(\lambda\ s:\ pp(s)\ \supset\ qq(s))(aa)\ \supset\ Inv(pp)(aa)\ \supset\ Inv(qq)(aa)$

$since\_since$: LEMMA
$Inv(\lambda\ s:\ |(\lambda\ s_0:\ |pp|(s_0)\ \leq\ rt1)|(s)\ \leq\ rt2\ \supset\ |pp|(s)\ \leq\ rt1\ +\ rt2)(aa)$

$since\_implies$: LEMMA
$Inv(\lambda\ s:\ pp(s)\ \supset\ qq(s))(aa)\ \supset\ Inv(\lambda\ s:\ |qq|(s)\ \leq\ |pp|(s))(aa)$

$min(rt1, rt2): time\ =\ ($IF $rt1\ \leq\ rt2$ THEN $rt1$ ELSE $rt2$ ENDIF$)$

$max(rt1, rt2): time\ =\ ($IF $rt1\ \leq\ rt2$ THEN $rt2$ ELSE $rt1$ ENDIF$)$

max_ax : LEMMA $rt1\ \leq\ max(rt1, rt2)\ \wedge\ rt2\ \leq\ max(rt1, rt2)$

min_ax : LEMMA $min(rt1, rt2)\ \leq\ rt1\ \wedge\ min(rt1, rt2)\ \leq\ rt2$

Some further relations between the *since* operator and the logical connectives are given in the theorem below.

$since\_or$: LEMMA
$Inv(\lambda\ s :\ |\lambda\ s_0 :\ pp(s_0)\ \vee\ qq(s_0)|(s)\ \leq\ min(|pp|(s), |qq|(s)))(aa)$

$since\_or2$: LEMMA
$Inv(\lambda\ s :$
$\quad |\lambda\ s_0 :\ pp(s_0)\ \vee\ qq(s_0)|(s)\ =\ |pp|(s)$
$\quad\quad \vee\ |\lambda\ s_0 :\ pp(s_0)\ \vee\ qq(s_0)|(s)\ =\ |qq|(s))(aa)$

$since\_ex\_middle$: LEMMA
$Inv(\lambda\ s :$
$\quad |\lambda\ s_0 :\ pp(s_0)\ \wedge\ qq(s_0)|(s)\ =\ |pp|(s)$
$\quad\quad \vee\ |\lambda\ s_0 :\ pp(s_0)\ \wedge\ \neg\ qq(s_0)|(s)\ =\ |pp|(s))(aa)$

$since\_and$: LEMMA
$Inv(\lambda\ s :\ max(|pp|(s), |qq|(s))\ \leq\ |\lambda\ s_0 :\ pp(s_0)\ \wedge\ qq(s_0)|(s))(aa)$

$since\_and1$: LEMMA $Inv(\lambda\ s :\ |pp|(s)\ \leq\ |\lambda\ s_0 :\ pp(s_0)\ \wedge\ qq(s_0)|(s))(aa)$

$since\_and2$: LEMMA $Inv(\lambda\ s :\ |qq|(s)\ \leq\ |\lambda\ s_0 :\ pp(s_0)\ \wedge\ qq(s_0)|(s))(aa)$

END $since\_thms$

5.3 Specifying the Railroad System

The railroad crossing system consists of three subsystems: the trains, the controller and the gate. The entire system is specified in the PVS theory *railroad* below. This theory is presented in five parts. The first part declares the types and program variables. The second part specifies the train subsystem. The third part specifies the controller. The fourth part specifies the behavior of the gate. The timing assumptions are given in the fifth and final part. The state of the train Tr in a state s is given by the predicate $train(Tr)(s)$. A train's state is either *safe*, *approaching* the crossing, or in the *crossing*. There are three possible actions for the subsystem consisting solely of the trains. A train can either go from a *safe* to an *approaching* state, from *approaching* to *crossing*, or from *crossing* to *safe* while leaving the state of any other trains unchanged. The train subsystem is *Approaching* (*Crossing*) if at least one train is *approaching* (*crossing*). It is in an *Unsafe* state if at least one train is either *approaching* or *crossing*.

The state of the controller *signal* is either *up* or *down*. The signal is set to *down* when a train is detected as approaching. It is set to *up* when no train is approaching or in the crossing.

The gate is either *up*, *down*, *moving_up*, or *moving_down*. The gate has two transitions that respond to the signal value by moving up or down, and two other transitions that take it from *moving_up* to *up*, and from *moving_down* to *down*.

$railroad$: THEORY
BEGIN

USING $time, since, since\_thms, compose$

$Train$: TYPE

$Tr, Tr1, Tr2, Tr3$: VAR $Train$

Tr_state : TYPE $= \{safe, approaching, crossing\}$

$train : [Train \to [state \to Tr\_state]]$

s, s_0, s_1 : VAR $state$

$Approaching(s) : bool = (\exists\ Tr : (train(Tr)(s) = approaching))$

$Crossing(s) : bool = (\exists\ Tr : (train(Tr)(s) = crossing))$

$Unsafe(s) : bool = (Approaching(s) \vee Crossing(s))$

$Safe(s) : bool = \neg\ Unsafe(s)$

$Gate$: TYPE $= \{up, down, moving\_up, moving\_down\}$

$gate : [state \to Gate]$

$Signal$: TYPE $= \{lower, raise\}$

$signal : [state \to Signal]$

The transitions of the train subsystem are given by *tr_action1*, *tr_action2*, and *tr_action3*, and the entire program for this component of the system is given by *tr_action*. The train subsystem expects its environment to leave the status of the trains unchanged.

$$
\begin{array}{l}
\mathit{tr\_action1} : \mathit{Action} \;=\\
\quad (\lambda\, s_0, s_1 :\\
\qquad (\exists\, \mathit{Tr} :\\
\qquad\quad \mathit{train}(\mathit{Tr})(s_0) \;=\; \mathit{safe}\\
\qquad\qquad \wedge\; \mathit{train}(\mathit{Tr})(s_1) \;=\; \mathit{approaching}\\
\qquad\qquad\qquad \wedge\; (\forall\, \mathit{Tr1} :\; \mathit{Tr1} \;\neq\; \mathit{Tr} \;\supset\; \mathit{train}(\mathit{Tr1})(s_1) \;=\; \mathit{train}(\mathit{Tr1})(s_0))))\\
\\
\mathit{tr\_action2} : \mathit{Action} \;=\\
\quad (\lambda\, s_0, s_1 :\\
\qquad (\exists\, \mathit{Tr} :\\
\qquad\quad \mathit{train}(\mathit{Tr})(s_0) \;=\; \mathit{approaching}\\
\qquad\qquad \wedge\; \mathit{train}(\mathit{Tr})(s_1) \;=\; \mathit{crossing}\\
\qquad\qquad\qquad \wedge\; (\forall\, \mathit{Tr1} :\; \mathit{Tr1} \;\neq\; \mathit{Tr} \;\supset\; \mathit{train}(\mathit{Tr1})(s_1) \;=\; \mathit{train}(\mathit{Tr1})(s_0))))\\
\\
\mathit{tr\_action3} : \mathit{Action} \;=\\
\quad (\lambda\, s_0, s_1 :\\
\qquad (\exists\, \mathit{Tr} :\\
\qquad\quad \mathit{train}(\mathit{Tr})(s_0) \;=\; \mathit{crossing}\\
\qquad\qquad \wedge\; \mathit{train}(\mathit{Tr})(s_1) \;=\; \mathit{safe}\\
\qquad\qquad\qquad \wedge\; (\forall\, \mathit{Tr1} :\; \mathit{Tr1} \;\neq\; \mathit{Tr} \;\supset\; \mathit{train}(\mathit{Tr1})(s_1) \;=\; \mathit{train}(\mathit{Tr1})(s_0))))\\
\\
\mathit{tr\_stutter} : \mathit{Action} \;=\; (\lambda\, s_0, s_1 :\; (\forall\, \mathit{Tr} :\; \mathit{train}(\mathit{Tr})(s_1) \;=\; \mathit{train}(\mathit{Tr})(s_0)))\\
\\
\mathit{tr\_action} : \mathit{Program} \;=\\
\quad (\#\; \mathit{init} := (\lambda\, s :\; (\forall\, \mathit{Tr} :\; \mathit{train}(\mathit{Tr})(s) \;=\; \mathit{safe})),\\
\qquad G := (\lambda\, s_0, s_1 :\; \mathit{tr\_action1}(s_0, s_1) \;\vee\; \mathit{tr\_action2}(s_0, s_1) \;\vee\; \mathit{tr\_action3}(s_0, s_1)),\\
\qquad R := \mathit{tr\_stutter}\#)
\end{array}
$$

Similarly, the controller component is shown below with the transitions *cont_action1* and *cont_action2*. The controller expects its environment to leave the value of the variable *signal* unchanged.

$$cont\_action1 : Action = (\lambda\ s_0, s_1 : Approaching(s_0) \wedge signal(s_1) = lower)$$

$$cont\_action2 : Action = (\lambda\ s_0, s_1 : Safe(s_0) \wedge signal(s_1) = raise)$$

$$cont\_stutter : Action = (\lambda\ s_0, s_1 : signal(s_1) = signal(s_0))$$

$$\begin{array}{l} cont\_action : Program = \\ \quad (\#\ init := (\lambda\ s : \text{TRUE}), \\ \qquad G := (\lambda\ s_0, s_1 : cont\_action1(s_0, s_1) \vee cont\_action2(s_0, s_1)), \\ \qquad R := cont\_stutter\#) \end{array}$$

The gate component has the four transitions shown below. The gate expects its environment to leave the value of *gate* unchanged.

$$\begin{array}{l} gate\_action1 : Action = \\ \quad (\lambda\ s_0, s_1 : signal(s_0) = lower \wedge gate(s_0) \neq down \wedge gate(s_1) = moving\_down) \end{array}$$

$$gate\_action2 : Action = (\lambda\ s_0, s_1 : gate(s_0) = moving\_down \wedge gate(s_1) = down)$$

$$\begin{array}{l} gate\_action3 : Action = \\ \quad (\lambda\ s_0, s_1 : signal(s_0) = raise \wedge gate(s_0) \neq up \wedge gate(s_1) = moving\_up) \end{array}$$

$$gate\_action4 : Action = (\lambda\ s_0, s_1 : gate(s_0) = moving\_up \wedge gate(s_1) = up)$$

$$gate\_stutter : Action = (\lambda\ s_0, s_1 : gate(s_1) = gate(s_0))$$

$$\begin{array}{l} gate\_action : Program = \\ \quad (\#\ init := (\lambda\ s : \text{TRUE}), \\ \qquad G := \\ \qquad\ (\lambda\ s_0, s_1 : \\ \qquad\quad gate\_action1(s_0, s_1) \\ \qquad\qquad \vee\ gate\_action2(s_0, s_1) \\ \qquad\qquad\quad \vee\ gate\_action3(s_0, s_1) \vee gate\_action4(s_0, s_1)), \\ \qquad R := gate\_stutter\#) \end{array}$$

There are five timing constraints given by the definitions of *TI0*, *TI1*, *TI2*, *TI3*, and *TI4*. *TI0* asserts that the initial value of $since(pp)$ exceeds the initial *Time* value. *TI1* asserts that *signal* is set to *lower* within D time units of a train approaching. *TI2* asserts that the gate is either *down* or *moving_down* within G time units of the controller's signal being set to *lower*. *TI3* asserts that the gate is down within L time units from when it starts moving down. *TI4* asserts that no train goes from *safe* to *crossing* in less than $D + G + L$ time units.

$i, j, k, m, n :$ VAR *nat*
$pp, qq, rr :$ VAR *statepred*

$aa, bb, cc :$ VAR *behavior*

$non\_negatives :$ TYPE $= \{x : rational \mid x \geq 0\}$ CONTAINING 0

$D, G, L : non\_negatives$

$TI0(aa) : bool = (\forall\ pp : |pp|(aa(0)) = Time(aa(0)) + 1)$

$$TI1(aa) : bool = Inv(\lambda\ s : Approaching(s) \wedge signal(s) \neq lower \supset |\lambda\ s_0 : \neg\ Approaching(s_0)|(s) \leq D)(aa)$$

$$TI2(aa) : bool = Inv(\lambda\ s : signal(s) = lower \supset gate(s) = down \vee gate(s) = moving\_down \vee |(\lambda\ s_0 : signal(s_0) \neq lower)|(s) \leq G)(aa)$$

$$TI3(aa) : bool = Inv(\lambda\ s : gate(s) = moving\_down \supset |(\lambda\ s_0 : gate(s_0) \neq moving\_down)|(s) \leq L)(aa)$$

$$TI4(aa) : bool = Inv(\lambda\ s : (\forall\ Tr : train(Tr)(s) = crossing \supset |\lambda\ s_0 : train(Tr)(s_0) = approaching|(s) + (D + G + L) < |\lambda\ s_0 : train(Tr)(s_0) = safe|(s)))(aa)$$

$$program(aa) : bool = aa \models (tr\_action \| (cont\_action \| gate\_action)) \wedge TI0(aa) \wedge TI1(aa) \wedge TI2(aa) \wedge TI3(aa) \wedge TI4(aa)$$

END *railroad*

5.4 The Safety Proof

The PVS theory *railroad_pr* contains the lemmas leading to the proof of the safety property. This theory is presented in five parts below. The first program invariant *Inv1* for the railroad crossing system asserts that the state when a train that is now approaching was last safe is at least as recent as the state when no train is approaching. Since this is a property that is purely about the train subsystem, we can use the theorem *deCompose1* to verify it as a property of the entire system by merely ensuring that it is preserved by the train transitions. This means that this invariant only has to be proved relative to the four train transitions, whereas in our previous non-compositional approach, this invariant had to be shown to be preserved by all of the ten system transitions.

$railroad\_pr$: THEORY
BEGIN

USING $time, railroad, since, since\_thms, compose$

$Tr, Tr1, Tr2, Tr3$: VAR $Train$

s, s_0, s_1 : VAR $state$

aa, bb, cc : VAR $behavior$

pp, qq, rr : VAR $statepred$

$rt, rt1, rt2, rt3$: VAR $time$

i, j, k, m, n : VAR nat

$Inv1$: LEMMA

$$
\begin{array}{l}
program(aa) \\
\quad \supset \\
\quad Inv(\lambda\ s : \\
\qquad\quad (\forall\ Tr : \\
\qquad\qquad train(Tr)(s)\ =\ approaching \\
\qquad\qquad\quad \supset\ |\lambda\ s_0 :\ train(Tr)(s_0)\ =\ safe|(s) \\
\qquad\qquad\qquad \leq\ |\lambda\ s_0 :\ \neg\ Approaching(s_0)|(s)))(aa)
\end{array}
$$

The second invariant *Inv2* uses *Inv1* and *TI1* to demonstrate that if a train *Tr* is approaching but the signal is not set to *lower*, then the time since *Tr* was last in its *safe* state does not exceed D.

Inv3 follows from *Inv1* and *TI2*. It is a minor adaptation of *TI2*.

$$
\begin{array}{l}
\mathit{Inv2} : \text{LEMMA} \\
\quad \mathit{program}(aa) \\
\qquad \supset \\
\qquad \mathit{Inv}(\lambda\ s : \\
\qquad\qquad (\forall\ \mathit{Tr} : \\
\qquad\qquad\quad \mathit{train}(\mathit{Tr})(s)\ =\ \mathit{approaching}\ \wedge\ \mathit{signal}(s)\ \neq\ \mathit{lower} \\
\qquad\qquad\qquad \supset\ |\lambda\ s_0 :\ \mathit{train}(\mathit{Tr})(s_0)\ =\ \mathit{safe}|(s)\ \leq\ D))(aa) \\
\\
\mathit{Inv3} : \text{LEMMA} \\
\quad \mathit{program}(aa) \\
\qquad \supset \\
\qquad \mathit{Inv}(\lambda\ s : \\
\qquad\qquad (\forall\ \mathit{Tr} : \\
\qquad\qquad\quad \mathit{train}(\mathit{Tr})(s)\ =\ \mathit{approaching}\ \wedge\ \mathit{signal}(s)\ =\ \mathit{lower} \\
\qquad\qquad\qquad \supset \\
\qquad\qquad\qquad (\mathit{gate}(s)\ =\ \mathit{down} \\
\qquad\qquad\qquad\quad \vee\ \mathit{gate}(s)\ =\ \mathit{moving\_down} \\
\qquad\qquad\qquad\qquad \vee\ |\lambda\ s_0 :\ \mathit{train}(\mathit{Tr})(s_0)\ =\ \mathit{safe}|(s)\ \leq\ (D\ +\ G))))(aa)
\end{array}
$$

The invariant in *Inv4_1* is shown to be preserved by the controller and gate subsystems together, and by their environment. It essentially asserts that once the signal is lower, the gate cannot make the transition from the *down* state to a non-*down* state. This proof thus considers three fewer transitions than the earlier non-compositional proof of the theorem.

Inv4_3 makes the trivial observation that the point at which the train goes from the *safe* to *approaching* state is the same as when it goes from any non-*approaching* state to an *approaching* state. This invariance is shown to be preserved by the train subsystem. This proof thus involves six fewer transitions than the previous attempt.

$$
\begin{array}{l}
\mathit{Inv4\_1} : \text{LEMMA} \\
\quad \mathit{program}(aa) \\
\qquad \supset \\
\quad \mathit{Inv}(\lambda\ s : \\
\qquad\qquad \mathit{signal}(s) = \mathit{lower} \wedge \mathit{gate}(s) \neq \mathit{down} \\
\qquad\qquad\quad \supset |\lambda\ s_0 : \mathit{signal}(s_0) \neq \mathit{lower}|(s) \\
\qquad\qquad\qquad\quad \leq |\lambda\ s_0 : \mathit{gate}(s_0) = \mathit{down}|(s))(aa)
\end{array}
$$

$$
\begin{array}{l}
\mathit{Inv4\_3} : \text{LEMMA} \\
\quad \mathit{program}(aa) \\
\qquad \supset \\
\quad \mathit{Inv}(\lambda\ s : \\
\qquad\qquad (\forall\ \mathit{Tr} : \\
\qquad\qquad\quad \mathit{train}(\mathit{Tr})(s) = \mathit{approaching} \\
\qquad\qquad\qquad \supset |\lambda\ s_0 : \mathit{train}(\mathit{Tr})(s_0) = \mathit{safe}|(s) \\
\qquad\qquad\qquad\qquad = |\lambda\ s_0 : \mathit{train}(\mathit{Tr})(s_0) \neq \mathit{approaching}|(s)))(aa)
\end{array}
$$

Inv4_4 is a variant of *Inv2*. *Inv4* restates the timing constraint *TI4* using *Inv3*, *TI3*, *Inv4_1*, *Inv4_3*, and *Inv4_4*.

$$
\begin{array}{l}
\mathit{Inv4\_4} : \text{LEMMA} \\
\quad \mathit{program}(aa) \\
\qquad \supset \\
\quad \mathit{Inv}(\lambda\ s : \\
\qquad\qquad (\forall\ \mathit{Tr} : \\
\qquad\qquad\quad \mathit{train}(\mathit{Tr})(s) = \mathit{approaching} \\
\qquad\qquad\qquad \supset |\lambda\ s_0 : \mathit{train}(\mathit{Tr})(s_0) = \mathit{safe}|(s) \\
\qquad\qquad\qquad\qquad \leq D + |\lambda\ s_0 : \mathit{signal}(s_0) \neq \mathit{lower}|(s)))(aa)
\end{array}
$$

$$
\begin{array}{l}
\mathit{Inv4} : \text{LEMMA} \\
\quad \mathit{program}(aa) \\
\qquad \supset \\
\quad \mathit{Inv}(\lambda\ s : \\
\qquad\qquad (\forall\ \mathit{Tr} : \\
\qquad\qquad\quad \mathit{train}(\mathit{Tr})(s) = \mathit{approaching} \wedge \mathit{gate}(s) \neq \mathit{down} \\
\qquad\qquad\qquad \supset |\lambda\ s_0 : \mathit{train}(\mathit{Tr})(s_0) = \mathit{safe}|(s) \leq (D + G + L)))(aa)
\end{array}
$$

The main safety property asserts the invariant that when the train is in its *crossing* state, the gate is *down* and the signal is *lower*. This is the only invariant that has to be proved relative to all the system transitions.

$$Safety\_pred(s) : bool = (Crossing(s) \supset gate(s) = down \wedge signal(s) = lower)$$

$$Safety : \text{LEMMA } program(aa) \supset Inv(Safety\_pred)(aa)$$

END *railroad_pr*

On the whole, the compositional approach roughly halves the effort involved in the proof. Since there is a combinatorial explosion of cases when a system is verified in a non-compositional manner, the savings from adopting a compositional approach are likely to be even larger when the systems considered are more complex.

6 Conclusion

Realistic concurrent and real-time systems are significantly more complex than those considered here. Existing verification techniques do not provide any effective ways to manage this complexity. We have described an approach to system composition that can be used to prove system properties by merely considering the subsystems that are relevant. The form of composition ensures that these properties will hold of the larger system. Our approach does not require the component specifications to be overly specific in order to demonstrate that a component respects the environment constraints of the other components. In specific cases where the component specifications are sufficiently detailed, it will be possible to demonstrate that any constraints placed on one component by another, are already met. In general, however, the burden of proving that components do not violate the constraints placed by the other components is shifted to the refinement phase. The two examples we have considered demonstrate that our approach leads to more succinct proofs with less constrained specifications.

Acknowledgements. The present report clearly owes a great deal to the work of Abadi, Lamport, Chandy, Misra, and Collette. The insightful comments of Connie Heitmeyer (Naval Research Laboratory), Ralph Jeffords (Locus, Inc.), and John Rushby (SRI) were extremely helpful. David Cyrluk and Sam Owre suggested several improvements to the presentation of this material. The project was funded by the US Naval Research Laboratory under contract N00015-92-C-2177.

References

[AL90] Martín Abadi and Leslie Lamport. Composing specifications. Technical Report 66, DEC Systems Research Center, Palo Alto, CA, October 1990.

[CM88] K. Mani Chandy and Jayadev Misra. *Parallel Program Design: A Foundation.* Addison-Wesley, Reading, MA, 1988.

[CMP94] E. Chang, Z. Manna, and A. Pnueli. Temporal specification and verification of reactive modules. *Journal of the ACM*, 1994. To appear.

[Col93] P. Collette. Application of the composition principle to Unity-like specifications. In M.-C. Gaudel and J.-P. Jouannaud, editors, *Proceedings of TAPSOFT '93*, volume 668 of *Lecture Notes in Computer Science*, pages 230–242, Berlin, 1993. Springer-Verlag.

[HJ93] C. L. Heitmeyer and R. D. Jeffords. Formal specification and verification of real-time system requirements: A comparison study. Technical report, Naval Research Laboratory, 1993. To appear.

[Mis90] J. Misra. Specifying concurrent objects as communicating processes. *Science of Computer Programming*, 14(2–3):159–184, 1990.

[OSR93] S. Owre, N. Shankar, and J. M. Rushby. *User Guide for the PVS Specification and Verification System, Language, and Proof Checker (Beta Release).* Computer Science Laboratory, SRI International, Menlo Park, CA, February 1993. Three volumes.

[Sha92] N. Shankar. Mechanized verification of real-time systems using PVS. Technical Report SRI-CSL-12, SRI International Computer Science Laboratory, Menlo Park, CA, 1992.

[Sha93] N. Shankar. Verification of real-time systems using PVS. In C. Courcoubetis, editor, *Proceedings of the Fifth International Conference on Computer-Aided Verification*, volume 697 of *Lecture Notes in Computer Science*, pages 280–291, Berlin, 1993. Springer-Verlag.