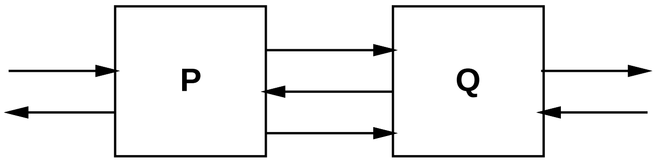
WEDNESDAY, Session 3
Compositional systems and methods

1 Introduction

We have already observed that composing finite state processes in parallel leads to an exponential explosion of states in the worst case, which imposes a strong limit on the size and complexity of a system that can be verified by state enumeration methods. Since most systems by design have a natural division into components or modules, it seems natural to try to reduce the number of processes by taking a divide-and-conquer approach. This means that we separate the specification of system into properties of its components, and verify the properties of the components separately. This of course leaves us the obligation of proving that the component specifications in turn imply the specification of the entire system. This is called compositional verification.

In the simplest case, imagine we have a system composed of two processes P and Q, which communicate with each other over a channel or channels, and also communicate with their environment:



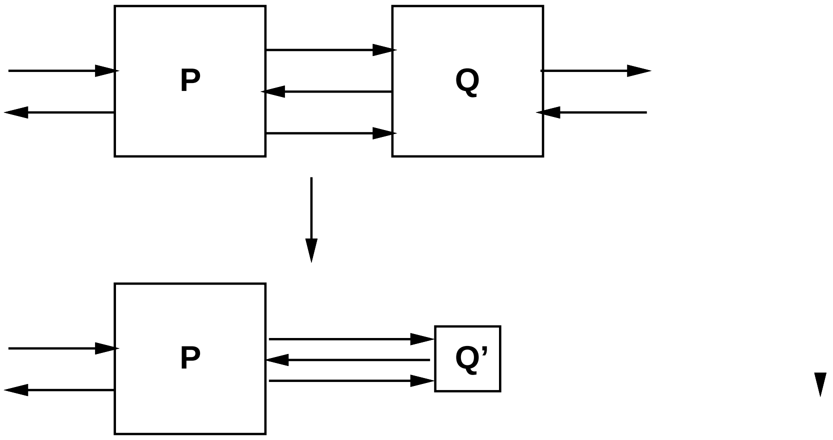
We denote this system $P \parallel Q$, for the parallel composition of P and Q. Given component specifications ϕ_P for process P and ϕ_Q for process Q, we would like to reason according to the following schema:

$$\frac{\begin{array}{c} P \models \phi_P \\ Q \models \phi_Q \\ \phi_P, \phi_Q \models \phi \end{array}}{P \parallel Q \models \phi}$$

Unfortunately, a component of a system is typically designed to work only in the environment of that system. Thus, the process P by itself will be unlikely to satisfy ϕ_P, or any other useful property in an arbitrary environment. Instead, the first unexpected input received by P will cuase it to enter an unintended part of its state space and thereafter to behave incorrectly. For this reason, the reachable state space of P in isolation may in fact be much larger than the state space of P composed with Q. We will call this the "environment problem", and consider two possible solutions:

1.0.1 Compositional minimization

Using this technique, we derive a "reduced" version Q' of Q' that characterizes just the behavior of Q that is visible to P via the channels connecting them:



This reduced environment Q' has been called an "interface process". Hiding the communications of Q that are not visible to P may allow us to reduce Q to an interface process Q' with a very small number of states. The interface process Q' may be obtained, for example, by automatically minimizing the finite automaton representing Q. This technique is called compositional minimization.

1.0.2 Assume/guarantee reasoning

In this case, we defer the environment problem by expressing any assumptions process P makes about process Q in a temporal formula ψ_Q. These assumptions can be used when checking that P satisfies its specification ϕ_P, and must, of course, be discharged relative to Q. To do this, we use an inference rule such as the following:

$$\frac{\begin{array}{c} P \parallel \psi_Q \models \phi_P \\ Q \models \psi_Q \end{array}}{P \parallel Q \models \phi_P}$$

In this lecture, we will lay out a general framework for compositional verification, and consider a specific model that fits into that framework.

2 A framework for compositional minimization

2.1 Framework

2.1.1 The interface rule

In this scheme we wish to apply an inference rule such as the following:

$$\frac{\begin{array}{c} Q' \equiv Q \downarrow \Sigma_P \\ \phi_p \in \mathcal{L}(\Sigma_P) \\ P \parallel Q' \models \phi_P \end{array}}{P \parallel Q \models \phi_P}$$

2.1.2 Elements of the framework

- Σ_P is the alphabet of process P (its inputs and outputs)
- $\mathcal{L}(\Sigma_P)$ is some logical language over Σ_P
- $\parallel$ is a compositinon operator on processes
- $\equiv$ is an equivalence relation on processes

2.1.3 Requirements

The interface rule can be applied in many specific process models and logics. In order for the interface rule to be valid in a particular framework, we have the following requirements:

1. **Congruence**: $\equiv$ must be a congruence w.r.t. $\parallel$. In other words

$$P \equiv P' \quad \wedge Q \equiv Q' \quad \Rightarrow P \parallel Q \equiv P' \parallel Q'$$

2. **Distributivity** of $\downarrow$ over $\parallel$: If $\Sigma \supseteq \Sigma_P$, we must have

$$P \parallel (Q \downarrow \Sigma) \equiv (P \parallel Q) \downarrow \Sigma$$

3. **Property preservation**: if $\phi \in \mathcal{L}(\Sigma)$, we must have
   - if $P \equiv Q$ and $P \models \phi$ then $Q \models \phi$

- if $P \downarrow \Sigma \models \phi$ then $P \models \phi$

The above requirements imply that

1. If $Q' \equiv Q \downarrow \Sigma_P$ then $P \parallel Q' \equiv (P \parallel Q) \downarrow \Sigma_P$.
2. Hence, if $P \parallel Q' \models \phi_P$, then $P \parallel Q \models \phi_P$.

In addition, since we want to obtain the interface process Q' automatically, we require an efficient minimization algorithm for $\equiv$. Ovbiously, we would also like to have an efficient model checking algorithm for $\mathcal{L}$. All these requirements strongly influnce the process model and logic we choose for compositional reasoning.

2.2 An example framework

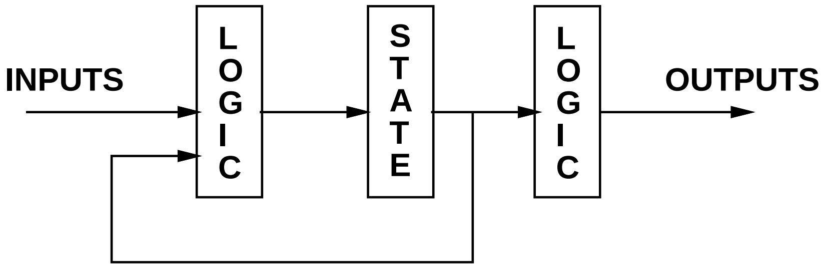
To illustrate the above concepts, consider the following instance of the compositional framework.

1. Processes are deterministic Moore machines
2. $\parallel$ is synchronous Moore machine composition
3. $\equiv$ is language equivalance
4. $\mathcal{L}$ is the logic CTL
5. The projection operator removes outputs of the Moore machines

Details on this system can be found in reference [1]. In the following, we consider the rationale behind these choices.

2.2.1 Moore machines

A Moore machine is a finite state machine where the outputs depend only on the state and not directly on the inputs:



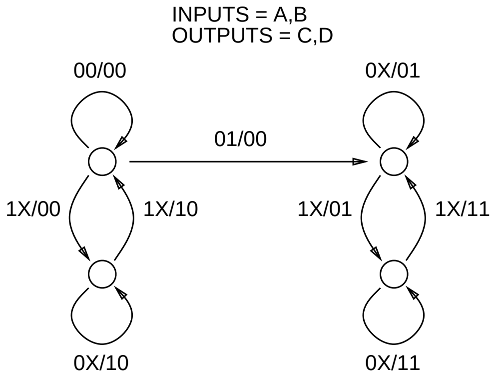
Deterministic machines

In a deterministic Moore machine:

- the current state and inputs uniquely determine the next state
- the current state uniquely determines the outputs

This guarantees exactly one output sequence for every input sequence.

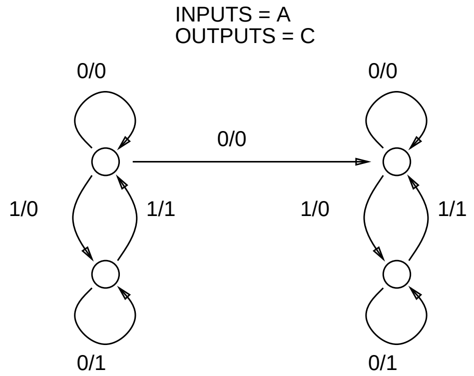
Deterministic models The *language* of a finite state machine is its set of input/output sequences. We model a Moore machine by a deterministic finite automaton that accepts this language:



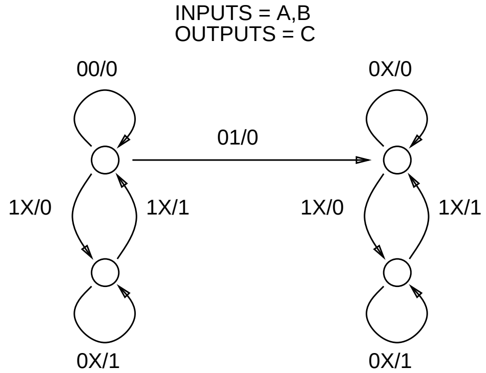
Note that we could model a deterministic state machine by a nondeterminstic automaton or *vice versa*. Using deterministic automata has some advantages and disadvantages, as we will see.

2.2.2 Projection and determinism

To compute $P \downarrow \Sigma$, for some more machine P and alphabet Σ, we simply erase those inputs and outputs not in Σ from the transitions. For the above example, this gives us the following automaton for $P \downarrow \{A, B\}$:

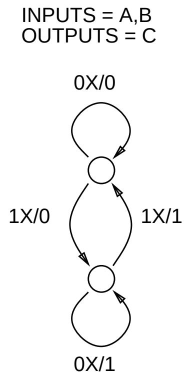


However, since erasing inputs may make the machine model nondeterministic, and the cost of determinizing is worst-case exponential, we will use a projection operator that erases only outputs. This gives, for the above example, a deterministic result:



2.2.3 Equivalence and minimization

We will say $P \equiv Q$ when the languages of P and Q are equal. Because our processes are deterministic, we can use the *nlogn* partition refinement algorithm of Hopcroft to minimize processes. The example above minimizes to:



Nondeterminism and strong bisimulation Note that in the nondeterministic case, there is no unique minimal automaton with a given langauge, and finding a (nonunique) minimal representative is a hard problem. However, we could use a stronger equivalence called "strong bisimulation".

- In the nondeterministic case strong bisimulation is the same as language equivalence.
- In the nondeterministic case, minimization w.r.t. strong bisimulation is $nlogn$ (Paige and Tarjan).

However, because bisimulation is a stronger equivalence, the reduction obtained may not be as great.

2.2.4 Composition operator

A synchronous composition $P \parallel Q$ of two Moore machines P and Q can be obtained by constructing a product of the two corresponding automata. This means that each state of the product consists of a pair of sates from P and Q respectively. The details of this are tedious. The important thing is that we have the two key properties:

- congruence of $\equiv$
- distribution of $\downarrow$ over $\parallel$

2.2.5 The logic CTL

To use the logic CTL in this framework, we require that:

- CTL properties over Σ are preserved by projection onto Σ.
- The logic must respect $\equiv$ (language equivalence).

The equivalence induced by CTL is strong bisimulation (that is, two states are indistinguishable by CTL formulas exactly when they are bisimilar).

Deterministic case: Since strong bisimulation and language equivalence are the same in this case, $\equiv$ preserves CTL formulas.

Nondeterministic case:

- Use bisimulation equivalence, *or*
- Use LTL (preserved by language equivalence, but harder model checking)

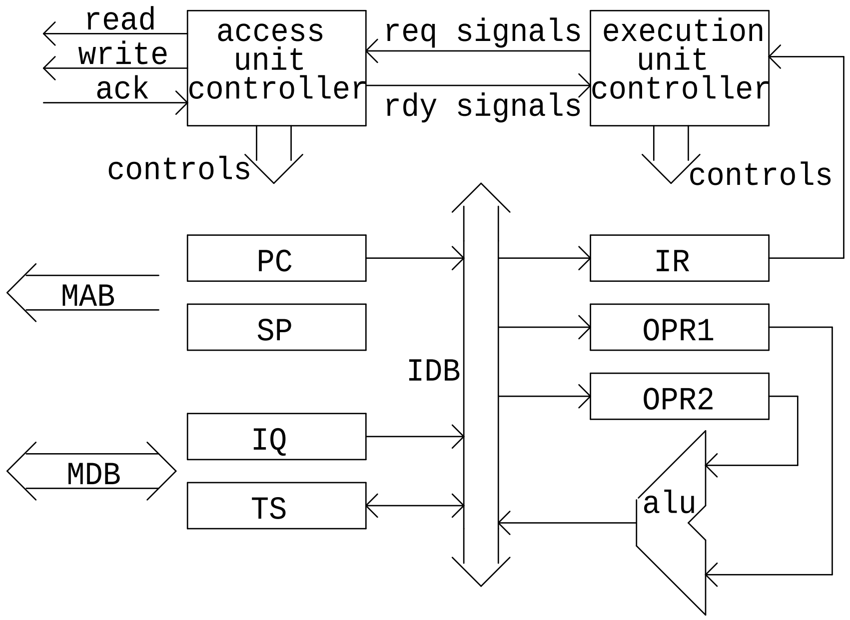
Note also that projection operators for asynchronous composition models generally change the length of the observed input/output trace by deleting unobservable actions. This means the properties in LTL and CTL are not preserved, and we must limit ourselves to logic subsets, like LTL-X (LTL without next-time).

2.3 Application: decoupled processor controller

As an example, we consider the finite-state control for a simple CPU with a stack-based intruction set. The processor is broken into two units:

- Access unit (AU) controller
  - Prefetches instructions
  - Caches top element of stack
  - Interfaces to memory unit
- Execution unit (EU) controller
  - Executes instructions

Both of these controllers are deterministic Moore machines, described in the simple procedural state machine language called SML (not "Standard ML"). For details about this example, see reference [2].



The four basic transactions that can occur between the execution unit and access unit are:

- Fetch next instruction
- Branch (set program counter)
- Push item onto stack
- Pop item off of stack

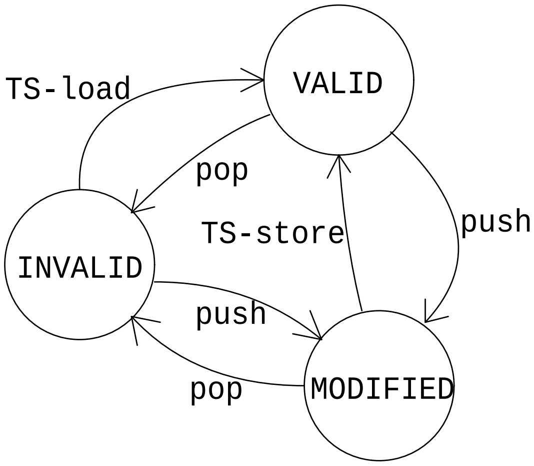
For each of these transaction a “reqest” signal passes from the EU to the AU and a “ready” signal passes from the AU to the EU.

Our verification strategy will be:

1. Write a set of temporal formulas about the AU controller that are sufficient to ensure correct management of the prefetch buffer and stack cache.
2. Hide the outputs of the EU controller that control the EU data path.
3. Minimize this process to produce an interface process for verifying the AU controller.

2.3.1 The AU controller specification

The first task is to provide temporal logic specifications for AU controller. We will consider only specifications for the stack-caching function. The stack cache itself can be in one of three possible states: INVALID, VALID and MODIFIED. The following diagram shows the operations allowable in each state and the resulting state changes:



We can translate these requirements into formulas of CTL by defining a macro operator:

$$(p \text{ BEFORE } q) \equiv (\neg q\ U\ p)$$

We specify the following:

- Pushed data must be stored to memory or popped before being overwritten:

$$AG(push \Rightarrow \neg EX((push \vee \textit{TS-load} \text{ BEFORE } (\textit{TS-store-done} \vee pop)))$$

- After popping the stack, the cache must be either loaded from memory or pushed before being accesses again:

$$AG(pop \Rightarrow \neg EX((pop \vee \textit{TS-store} \text{ BEFORE } (\textit{TS-load-done} \vee push)))$$

- Initially, the cache must be filled before being accessed:

$$\neg EX((pop \vee \textit{TS-store} \text{ BEFORE } (\textit{TS-load-done} \vee push)))$$

Additional properties specify:

- When busses may be driven and registers written
- Stability of address and data during memory read and write
- Prefetch operation

Finally, there is also a liveness requirement:

$$AG\ AF\ fetch$$

2.3.2 Compositional verification results

- AU controller alone: 13 states
  - Some properties not satisfied
  - Properties that are satisfied are not necessarily preserved in composition
- AU $\|$ EU: 1274 states
- AU $\|$ EU$'$: 221 states

Reduction in size of interface process is due to fact that, all though there are many defferent instructions, these fall into only a small number of memory access patterns.

Advantages of compositional minimization approach:

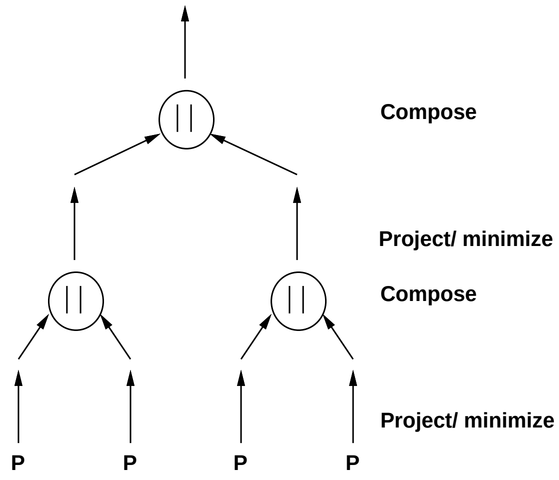
- Reduction is completely automatic
- No abstraction is done
  - Results are exact (no false negatives)
  - Counterexamples are always valid

Disadvantages:

- No abstraction is done
  - Not much reduction is available, since interface process must be equivalent to original w.r.t. interface signals.
- Must still prove that decomposed specifications imply overall specification.

2.4 Hierarchical minimization

In a hierarchically structured system, we can miminize before composition in a hierarchical way:



Advantage:

- Complex systems sometimes reduce to very small processes when view through limited inputs/outputs (e.g., a channel)

Disadvantage:

- Environment problem: intermediate processes may be much larger than full composition due to unreachable states. This usually requires some manual intervention to identify "failure" states.

For an example of hierarchical minimization in practice, see reference [3].

3 Assume/guarantee style reasoning

As stated previously, in the assume/guarantee paradigm, we use a temporal formula ψ to represent any assumptions that process P might make about process Q. We can then apply an inference rule such as the following to show that P satisfies its specification ϕ:

$$\frac{\begin{array}{c} P \parallel \psi \models \phi \\ Q \models \psi \end{array}}{P \parallel Q \models \phi}$$

Note that by $P \parallel \psi \models \phi$ we mean that $P \parallel Q' \models \phi$ for all Q' such that $Q' \models \psi$. This method is more flexible than compositonal minimization, but also requires more human input.

3.1 Framework

In this framework, we have the following elements:

1. A preorder $\leq$ on processes (transitive relation)
2. Parallel composition operator $\|$
3. A logic $\mathcal{L}$.

3.1.1 Requirements

The assume/guarantee rule can also be applied in many specific process models and logics. In order for it to be valid in a particular framework, we have the following requirements:

1. **Monotonicity**: Composition operator must be monotonic w.r.t. $\leq$. That is,

$$P \leq P' \wedge Q \leq Q' \;\Rightarrow\; (P \parallel Q) \leq (P' \parallel Q')$$

2. **Property preservation**: for formulas $\phi \in \mathcal{L}$,

   $P \leq P' \wedge P' \models \phi$ implies $P \models \phi$

In addition, in order to check assertions of the form $P \parallel \psi \models \phi$, we need the following:

> **Maximal model property**: for all formulas $\psi \in \mathcal{L}$ there exists a process T_ψ such that
> $$P \leq T_\psi \quad \text{iff} \quad P \models \psi$$

Given the maximal model property, $P \parallel \psi \models \phi$ becomes equivalent to $P \parallel T_\psi \models \phi$, which we verify by model checking. The complexity of this depends on the complexity of the model checking problem and the construction of T_ψ, which in turn affects our choice of process model and logic.

The above requirements imply that

1. If $Q \models \psi$ then $Q \leq T_\psi$.
2. Hence if $P \parallel T_\psi \models \phi$ then by monotonicity, $P \parallel Q \models \phi$.

3.2 An example framework

To illustrate the above concepts, consider the following instance of the compositional framework.

1. Processes are deterministic Moore machines
2. $\|$ is synchronous Moore machine composition
3. $\leq$ is the "fair simulation" preorder
4. $\mathcal{L}$ is the logic ACTL

Details on this system can be found in reference [4].

3.2.1 The restricted logic ACTL

ACTL is the subset of CTL containing just those formulas which have no E path quantifiers when put in positive normal form. Thus, for example:

- $AG(p \Rightarrow \neg EXq)$ is in ACTL, since in PNF it is $AG(\neg p \vee AX\neg q)$.
- $AG\neg AFq$ is not in ACTL, since in PNF it is $AGEG\neg q$.

3.3 Why not ordinary CTL?

In general the maximal models for CTL formulas are tree automata and not word automata, which makes the compositional model checking problem more complex. As we will see, however, an ACTL formula has a maximal model with respect to the simulation preorder that is an ordinary word automaton, with "fairness constraints" (conditions that must happen infinitely often).

3.3.1 Simulation preorder and the tableau construction

Fair simulation is a relation between processes (automata) with fairness constraints that is

- equivalent to language containment in the deterministic case.

- stronger than language containment in the nondeterministic case.

Thus, in this framework, it is always true that if $P \leq P'$ then the language of P is contained in the language of P'. The exact definition of fair simulation is too tedious to enter into here. However, from our point of view the important facts are:

- If $P \leq P'$, then P satisfies all ACTL properties satisfied by P'.
- Any ACTL formula ψ has a maximal model T_ψ of size $O(2^{|\psi|})$ (with $O(|\psi|)$ fairness constraints) which can be obtained by a "tableau construction".

3.3.2 Complexity of compositional verification

The basic problem is to verify statements of the form

$$P \parallel \psi \models \phi$$

In the current framework, we can do this by substituting the tableau automaton T_ψ for ψ and model checking ϕ under fairness. This process is

- Linear in ther size of P times the size of ϕ.
- Singly exponential in the size of ψ.

3.4 Application example – CPU controller revisited

Instead of using an iterface process to stand for the EU controller while verifying the AU controller, we can instead replace the EU controller automaton by a simple ACTL formula, by means of the tableau contruction. As an example, it turns out that the property:

$$\phi = AG(push \Rightarrow \neg EX((push \vee \textit{TS-load} \text{ BEFORE } (\textit{TS-store-done} \vee pop)))$$

can be verified under just the assumption that the EU does not attempt a push and a pop operation at the same time:

$$\psi = AG\neg(push \wedge pop)$$

In practice, these conditions are usually found by first checking ϕ under no assumptions, obtaining a counterexample, and then adding an assumption to ψ that rules out this counterexample. Our proof of the given property ϕ now looks like the following:

$$\frac{\begin{array}{c}\mathrm{AU} \parallel \psi \models \phi \\ \mathrm{EU} \models \psi\end{array}}{\mathrm{AU} \parallel \mathrm{EU} \models \phi}$$

Since T_ψ is considerably smaller than the EU controller itself, we have simplified the model checking problem by compositional reasoning.

As a second example, the liveness property

$$\phi = AG\ AF\ fetch$$

requires us to reason in multiple steps. The first step is to use the following assumptions about the EU:

- ψ_1 : No two operations attemped at the same time
- ψ_2 : $AG(push\text{-}req \Rightarrow A(push\text{-}req\ W\ push\text{-}rdy))$
- ψ_3 : $AG(pop\text{-}req \Rightarrow A(push\text{-}req\ W\ pop\text{-}rdy))$

to show the following about the AU:

- ψ_4 : $AG(push\text{-}req \Rightarrow A(push\text{-}req\ U\ push\text{-}rdy))$
- ψ_5 : $AG(pop\text{-}req \Rightarrow A(push\text{-}req\ U\ pop\text{-}rdy))$

Essentially, we are showing that if the EU is persistent, a push or pop operation will always eventually complete. We also show about the AU controller that if branch operations do not occur infinitely, the fetch queue will eventually be filled:

- ψ_6 : $AG\ AF\ (fetch\text{-}rdy \vee branch)$

Since the EU always does only a finite number of pushes, pops and branches between fetches, these last three properties can be used to show that an instruction is always eventually fetched. Each of the above assertions was arrived at by examining the counterexample obtained when attempting to prove another assertion. Our overall proof looks like the following:

$$\frac{\begin{array}{c}\mathrm{EU} \models \psi_1 \wedge \psi_2 \wedge \psi_3 \\ \mathrm{AU} \parallel (\psi_1 \wedge \psi_2 \wedge \psi_3) \models \psi_4 \wedge \psi_5 \\ \mathrm{AU} \models \psi_6 \\ (\psi_4 \wedge \psi_5 \wedge \psi_6) \parallel \mathrm{EU} \models \phi\end{array}}{\mathrm{AU} \parallel \mathrm{EU} \models \phi}$$

If nothing else, this shows the complexity of constructing modular liveness proofs in the case when control passes back and forth between modules in complex ways. As mentioned above, automatically generated counterexamples do provide some guidance in constructing the proof. In general, however, there is no guarantee that the process of generating environment assumptions from counterexamples will eventually converge!

4 Conclusion

- Compositional methods manage the complexity of verifying large systems by divide-and-conquer.
- This leads to the "environment problem", since processes generally satisfy no useful properties out of context. Possible solutions:
  - Compositional minimization – automatically generate environment by projection and minimization.
  - Assume/guarantee reasoning – express environment assumptions in temporal logic.
- In some sense, these techniques only postpone the environment problem:
  - Reachable states problem in compositional minimization.
  - Unbounded regression problem in assume/guarantee reasoning – assumptions must be discharged realtive to other assumptions, *ad infinitum.*
- Remains to be seen if these techniques are really useful.

References

[1] E. M. Clarke, D. E. Long, and K. L. McMillan. Compositional model checking. In *Proceedings of the Fourth Annual IEEE Symposium on Logic in Computer Science*, 1989.

[2] E. M. Clarke, D. E. Long, and K. L. McMillan. A language for compositional specification and verification of finite state hardware controllers. In *9th International Symposium on Hardware Description Languages and their Applications*, 1989.

[3] W.M. Elseaidy, R. Cleaveland, and J.W. Baugh Jr. Modeling and verifying active structural control systems.

[4] O. Grumberg and D.E. Long. Model checking and modular verification. *ACM Trans. on Prog. Lang. and Syst.*, 16(3):843–871, May 1994.