Interface and component-based design for heterogeneous systems

Roberto Passerone

Department of Information and Communication Technology
University of Trento
Trento, Italy

Outline

- Introduction
  - Motivations
  - Interfaces and Components
  - Compatibility and Refinement
- Theory
  - Component and interface models
  - Refinement
  - Conformance orders and compatibility sets
  - Mirrors
- Behavior Compatibility
  - Closed and open systems

Motivations

- Reuse strategy critical for cost and time-to-market
- Systems assembled from internal and third party IPs
- Correctness of composition must be verified
  - Costly simulations may still miss problems
  - Safety-critical applications require a formal correctness proof
- Abstract component models used to specify the requirements
  - Transaction-level models shorten time to verification
  - Standards used to simplify the verification problem
- Formal proofs usually based on type systems
  - Typically only limited to static information

Extending types to richer models

- Define the rules of interaction
  - Static typing
  - Dynamic behaviors
  - Non-functional performance parameters
- Distinguish between roles and responsibilities
  - Assumptions on inputs accepted from the environment
  - Guarantees on the outputs generated
- Leads to a notion of compatibility
  - Informally, the output guarantees must satisfy the input assumptions

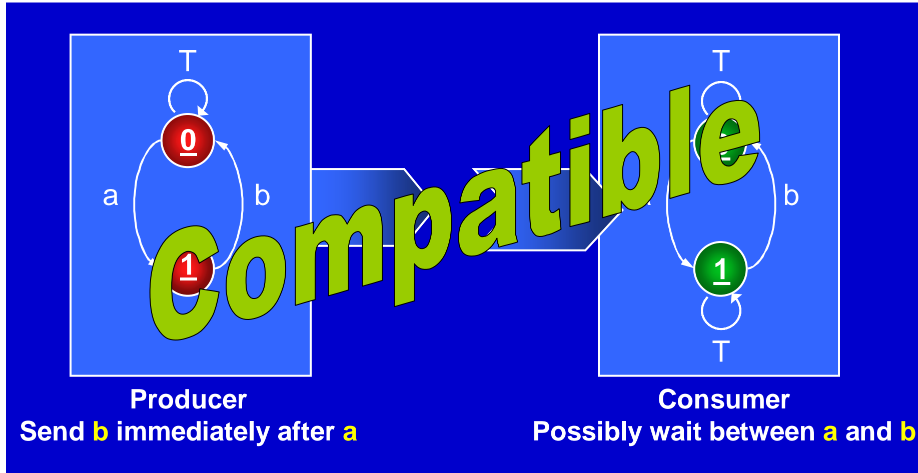
Component models

- Descriptive
  - What does the component do?
  - Examples: a C++ class definition (modulo static typing), a Verilog module, a device simulation model
- Defines behavior under all possible inputs
  - Terminology: input enabling, progressive, receptive
  - Does not constrain the environment
- Composes well under any environment
  - Component *will* do something in the end
  - Appropriate for describing implementations

Interface models

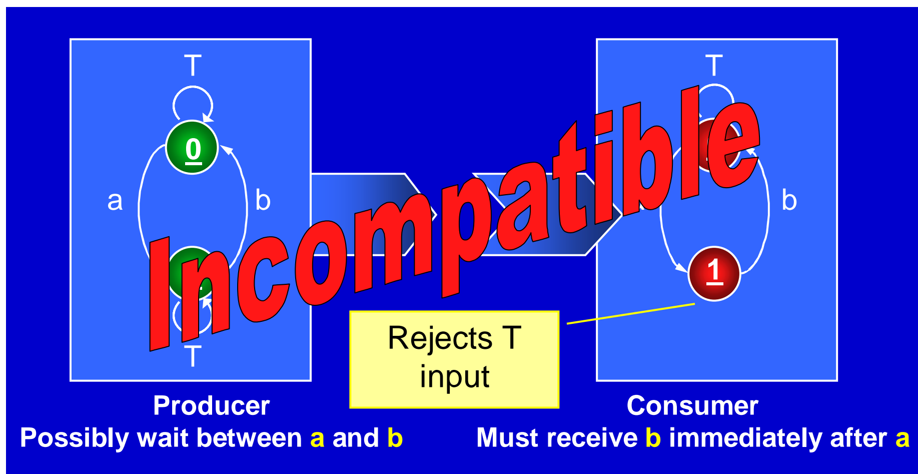
- Prescriptive
  - How can a component be combined with other components?
  - Examples: typing of a C function, Java interface definition, I/O signature of a Verilog module, operating conditions of a device
- Defines requirements under which a component may be used
  - Not all inputs are legal
  - Constrains the environment to avoid those inputs
- May not compose under some environment
  - Rejects environments that violate assumptions
  - Appropriate for describing specifications

Example



Data partitioned into two parts: a and b

Example revisited



Data partitioned into two parts: a and b

Observations

- Compatibility depends on who controls the signals
  - The input/output profile is essential
- Compatibility here reduced to checking behavior containment
  - Producer Outputs $\subseteq$ Consumer Inputs
- Symbols are used to represent data
  - Data must be represented explicitly when the behavior depends on the values
- Some mechanism in the implementation must signal whether a or b is being transferred
  - We don't need to be specific at this level of description
  - Any mechanism will do (toggling bits, additional signal, etc.)

Uses of interfaces

- Document a design
  - Provides formal specification to which components must comply
- Compatibility checking and assumption propagation under composition
  - Incremental design
- Interface refinement checking
  - Compliance between interfaces or between interface and component
- Compositional verification
  - Essential for efficiency
- Synthesis of interface requirements
  - Derive requirements on yet unknown parts of the system

Objectives

- Informally introduced two notions of consistency
  - Compatibility of assumptions and guarantees of interface models in a composition
  - Refinement or implementation relation between interfaces, or between an interface and a component
- The two notions are not independent
  - A consistent refinement must be compatible in the context of its specification
- We seek a general way to define and reason about them
  - Derive a formalism that is independent of the particular model of computation
  - Essential for heterogeneous models

Outline

- Theory
  - Define component and interface models
  - Define a notion of refinement as a function of a compatibility condition (conformance)
  - Derive their relation with the operation of composition (mirrors)

Domain of computation

- A set of computational blocks, called the *agents*
  - The automata in a finite state model
  - The Verilog modules in a discrete event model
  - The data-flow actors
- Each agent is characterized by the set of “signals” it uses
  - Signals may be values, actions, events, etc.
  - We call the set of signals the *alphabet A* of the agent
- A set of operators on agents
  - scoping: proj(B)(p)
  - instantiation: rename(r)(p)
  - composition: $p_1 \;||\; p_2$

Framework algebraic structure

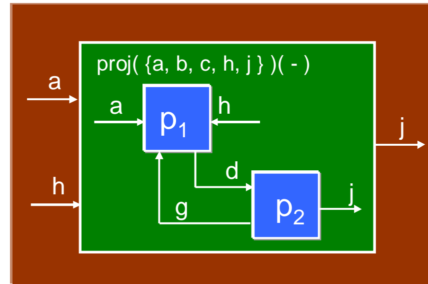
- Different models fit this template
  - We can reason abstractly without saying what the operators actually do
  - Our results will be valid for all models of computation
  - Our model has the structure of an algebra
- To comply with their intuitive interpretation, the operators must satisfy certain properties (axioms), e.g.
  - Parallel composition is associative and commutative
  - rename(id)(p) = p
  - proj(A)(p) = p
  - etc.

Input/Output model

Each agent simply consists of a set of input ports and a set of output ports

Agents are of the form

$p = (I, O)$ where I and O are disjoint sets of signal names and $A = I \cup O$



$\text{proj}(B)(p) = (I \cap B, O \cap B)$ iff $I \subseteq B$

$\text{rename}(r)(p) = (r(I), r(O))$ iff $A \subseteq \text{dom}(r)$

$p_1 \,||\, p_2 = ((I_1 \cup I_2) - (O_1 \cup O_2), O_1 \cup O_2)$ iff $O_1 \cap O_2 = \varnothing$

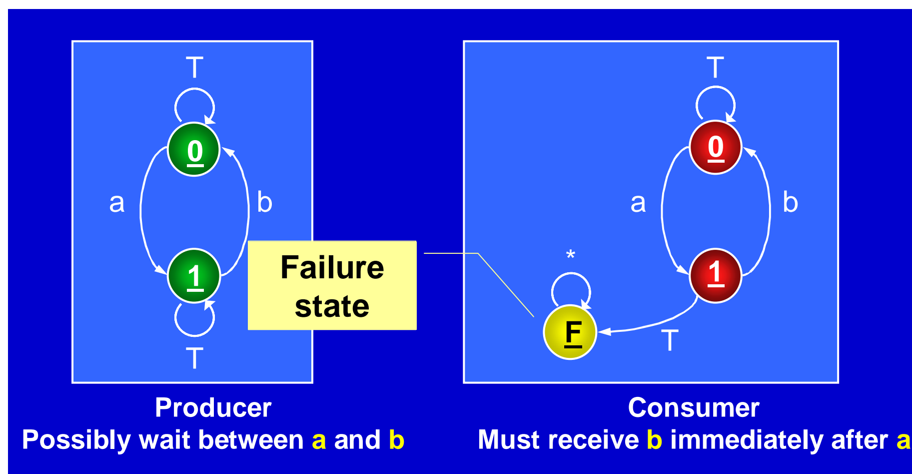
Synchronous component model

- Let A = { a, b, c } be a set of signals
  - Behaviors are elements of the language $\mathcal{L}_A = (2^A)^*$, e.g.
  - x = < {a}, {b}, {bc}, ∅, {ab}, ∅, {b}, … >
  - $p = (A, P)$ where $P \subseteq (2^A)^*$
- Projection operator
  - Set intersection
  - proj({a, c})(x) = < {a}, ∅, {c}, ∅, {a}, ∅, ∅, … >
- Parallel composition operator
  - $p_1 \,||\, p_2 = (A = A_1 \cup A_2, \text{proj}^{-1}(A)(P_1) \cap \text{proj}^{-1}(A)(P_2))$
  - A behavior is part of the composite if and only if it is a behavior of each of the components
  - Composition here is synchronous, because ∅ keeps track of time when there are no events

Synchronous interface model

- Each agent includes a set of *successes* and a set of *failures*
  - $p = (A, S, F); \quad P = S \cup F$
  - The failures represent possible behaviors that are illegal uses of the component
  - Assumes environment behaves only in $\neg F$
  - Interesting, for example, for protocol specification
- Composition
  - $p_1 \,||\, p_2 = (A_1 \cup A_2, S_1 \cap S_2, (F_1 \cap P_2) \cup (F_2 \cap P_1))$
  - To simplify, ignored inverse projection here
  - A behavior is a failure of the composite if and only if
    - It is a failure of one of the components
    - It is a possible behavior of the other component
  - Otherwise the failure is ruled out by the other component

Example revisited



Data partitioned into two parts: a and b

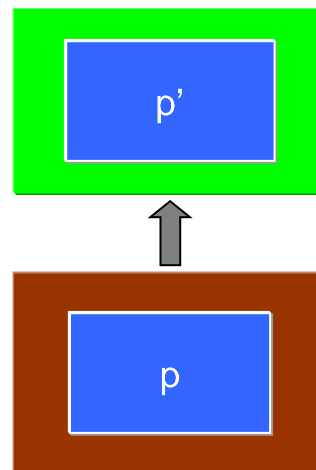
Compatibility and refinement

- Failure represent assumptions on the environment
  - For compatibility, the other components must not excite the failure behavior
- Compatibility reduces to checking the presence of failure behaviors
  - Two interfaces are compatible if their composition is “failure-free”
- We want to explore the relation between compatibility and refinement
  - We will define the refinement order in terms of compatibility
  - The generic definition can be applied to interface, as well as to component models
  - Interface and component models thus differ only in their level of abstraction

Refinement verification

- Have a specification *p’*
- Have an implementation *p*
- Want to show that *p* refines *p’*
- Refinement

  p can be substituted for *p’* in every context where *p’* can occur



Refinement and substitutability

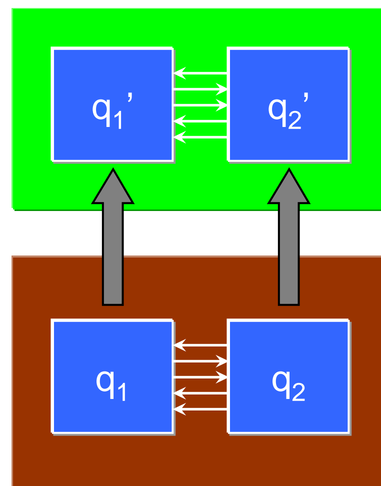
- A (pre)ordered agent algebra includes a preorder $\leq$ on the agents
- One possible interpretation of the order is substitutability
  - $p \leq p'$ if and only if p can be substituted for p'
- Or refinement
  - $p \leq p'$ if and only if p refines p'
- **Principle.** We require that in an ordered agent algebra the operators be monotonic with respect to the order
  - If f is an operator, then f is monotonic if and only if for all p and p', if $p \leq p'$ then $f(p) \leq f(p')$
- Monotonicity is essential to apply compositional methods
  - If $p \leq p'$ and $q \leq q'$ then $p \,||\, q \leq p' \,||\, q'$

Compositional verification

- Specification $p' = q_1' \,||\, q_2'$
- Implementation $p = q_1 \,||\, q_2$
- Want to show that p refines p'
- Compositionality

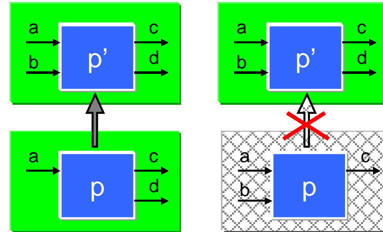
  Assume $q_1 \leq q_1'$ and $q_2 \leq q_2'$

  Then $p \leq p'$



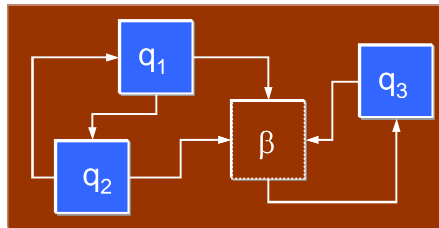
Substitutability for I/O agents

- Recall the definition of I/O agents
  - $p = (I, O)$
  - $p' = (I', O')$
- Define the following order on agents
  - $p \leq p'$ if and only if $O = O'$ and $I \subseteq I'$
- Other orders are also possible
  - We have defined the weakest (strongest?) order such that the operators are monotonic



Contexts

- The implementation makes use of a hierarchical structure
  - Expressions in the algebra represent the structure
  - *proj(A)(p || rename(r)(q)) || rename(r_2)(proj(B)(q_2))*
- A context is an expression *E* with one free variable β
  - The variable represents the "hole" in the structure



Compatibility

- Two agents p and q may be incompatible
  - Roughly speaking, *p* and *q* are compatible when their parallel composition is defined (and failure free)
- No matter what compatibility is, the following must be true
  - If *p'* is compatible in a certain context, and if $p \preceq p'$, then *p* must be compatible in the same context
  - In fact, this must be true for all contexts in which *p'* is compatible
- **Decision.** Compatibility becomes a parameter of our model
  - We express compatibility in terms of a set *G* of agents, called conformance set
  - *p'* is compatible in *E* iff $E[p'] \in G$
  - Similarly, *p* and *q* are compatible iff $p \| q \in G$

Conformance order

- G induces an order called G-conformance order
  - $p \preceq_G p'$ if and only if for all contexts *E*, if $E[p'] \in G$ then $E[p] \in G$
- If the order $\preceq$ in *Q* is the same as the order $\preceq_G$ induced by *G*, we say that *Q* has a *G*-conformance order
  - Note: different *G*'s induce different conformance orders
- For the I/O agents
  - Let *G* be the set of agents that have no inputs
- Then
  - The I/O agent algebra has a *G*-conformance order

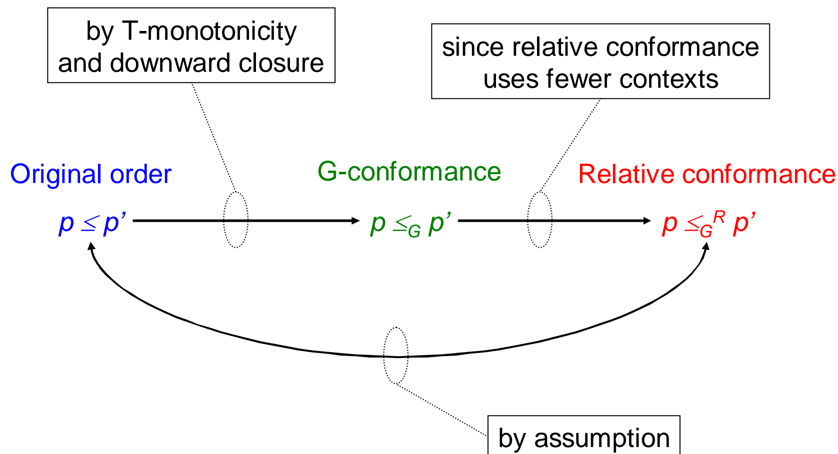
Relative conformance

- If Q has a G-conformance order, we can check refinement by checking conformance
  - But complexity is high, since it involves considering all possible compatible contexts
- Generalize conformance to only a subset of the contexts
  - $p \preceq_G^R p'$ if and only if for all contexts $E \in R$, if $E[p'] \in G$ then $E[p] \in G$
  - We call this relative conformance
  - Checking relative conformance is less complex than checking conformance

Relative conformance

- **Theorem.** If Q has a G-relative conformance order, then Q has a G-conformance order
  - Non trivial result: intuition goes the other way around!
- We are particularly interested in composition contexts
  - We call it conformance relative to composition
  - $p \preceq_G p'$ if and only if for all q, if $p' \| q \in G$ then $p \| q \in G$
- For the I/O agents
  - Let G be the set of agents that have no inputs
- Then
  - The I/O agent algebra has a G-conformance order relative to composition

Proof of theorem



Compatibility set

- Only certain composition contexts are relevant!
- Compatibility set
  - $cmp(p') = \{ q \mid p' \| q \in G \}$
- **Theorem.** If Q has a G-conformance order relative to composition, then
  - $p \preceq p'$ if and only if $p \| cmp(p') \subseteq G$
- Restrict the compatibility set further to just the maximal element
  - $mcp(p') = maximal(cmp(p'))$
- **Theorem.** If Q has a G-conformance order relative to composition, then
  - $p \preceq p'$ if and only if $p \| mcp(p') \subseteq G$
- **Result.** Conformance relative to composition can be checked by only considering the maximally compatible agents

Mirrors

- The greatest element of the compatibility set of an agent *p'* is called its *mirror*
  - It contains all the information of the remaining compatible agents
  - It can be used to check refinement
- A mirror function is a map from *D* to *D* such that
  - $p \leq p'$ if and only if $p \,||\, mirror(p') \in G$
  - Mirrors exist if and only if Q has a *G*-conformance order relative to composition and for all agents *p', cmp(p')* has a greatest element
- Mirror properties (similar to complementation)
  - $p = mirror^2(p)$
  - $p \leq q$ if and only if $mirror(q) \leq mirror(p)$

Synchronous component model

- Choose *G* to be the set of all agents with empty set of behaviors
- Refinement and conformance order
  - $p_1 \leq p_2$ if and only if $P_1 \subseteq P_2$
- Mirror
  - mirror(p) = ($A, \mathcal{L}_A - P$)
- A component model gives only guarantees
  - It guarantees the component has no behaviors outside *P*
  - The refinement provides stronger guarantees than its specification (fewer behaviors are possible)
  - Thus it can be used in place of its specification

Synchronous interface model

- Choose *G* to be the set of all agents with empty set of failures
- Refinement and conformance order
  - $p \leq p'$ if and only if $F \subseteq F'$ and $P \subseteq P'$
- Mirror
  - mirror(p) = (A, $S - F$, $\mathcal{L}_A - P$)
- An interface model provides assumptions as well as guarantees
  - The environment is not supposed to excite the behaviors in F
  - The refinement provides stronger guarantees than its specification (fewer behaviors are possible), and requires weaker assumptions (there are fewer failures)
  - Thus it can be used in place of its specification

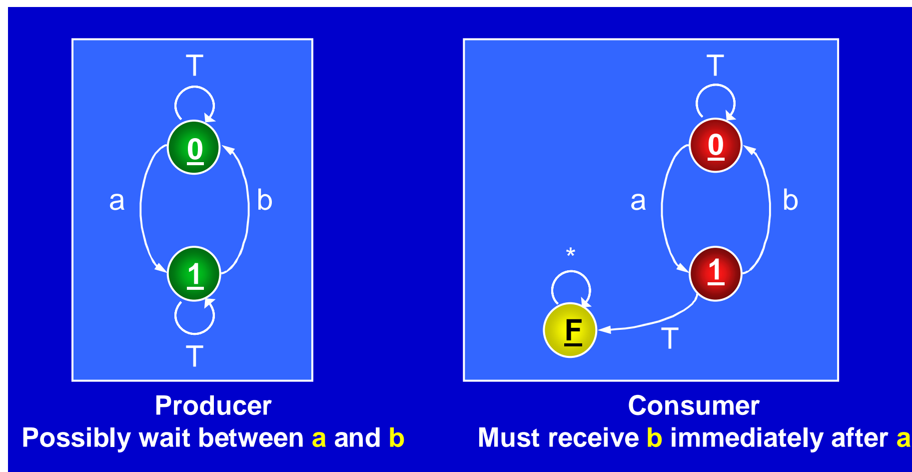
Assumption/guarantee propagation

- Composition computes assumptions and guarantees for the composite
  - $p_1 \| p_2 = (A_1 \cup A_2, S_1 \cap S_2, (F_1 \cap P_2) \cup (F_2 \cap P_1))$
- Let
  - $R = \neg F$ (required behaviors)
  - $G = S \cup F$ (guaranteed behaviors)
- Then
  - $R = (R_1 \cap R_2) \cup \neg G_1 \cup \neg G_2$
  - $G = G_1 \cap G_2$
- The interaction discharges some of the assumptions
  - Assumptions are propagated to the composite, but relaxed
  - Behaviors that are ruled out by the composition become acceptable, since they will not occur anyway

Compatibility of interface models

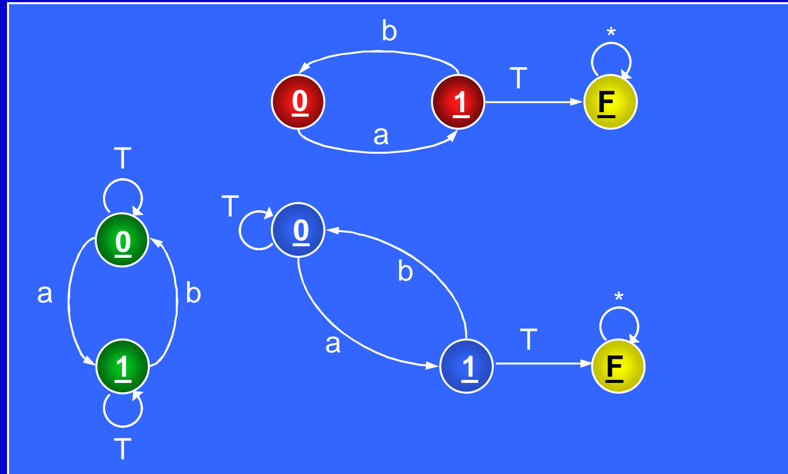
- *G* is the set of failure-free agents
- For a closed system compatibility is easy
  - Two components are compatible if their composition is failure-free
- For open systems compatibility is more subtle
  - There may still be failures in the composition
  - But there could be environments that prevent the failure to be excited
  - The environment must be capable of controlling the composition
- The input/output profile is therefore essential
  - A port is an input if the agent is receptive relative to that port
  - It is ready to accept any value

Example revisited

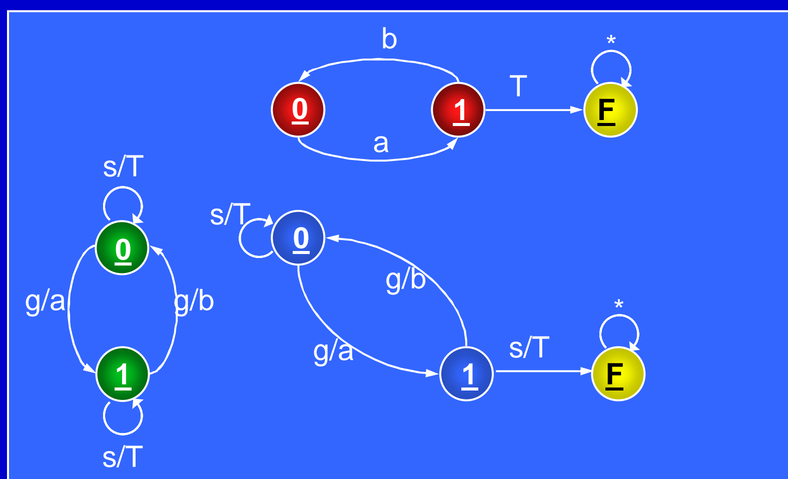


Data partitioned into two parts: a and b

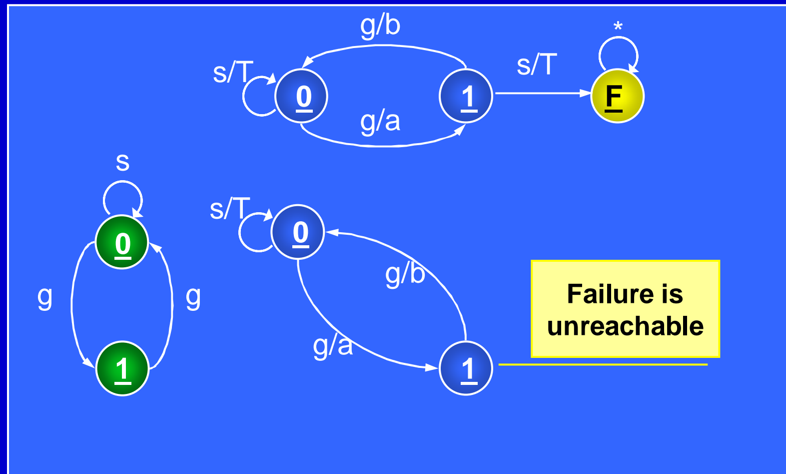
Example revisited: closed system



Example revisited: open system



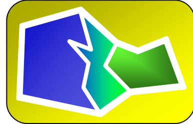
Helpful environment



Conclusions

- Discussed about component and interface models
  - Interfaces provide assumptions and guarantees
  - Essential for incremental design and component reuse
  - Components can be seen as interfaces with no assumptions
  - They differ in their level of abstraction
- Derived a relation between compatibility and refinement
  - Developed a model independent formalism
  - Formulated refinement order as a compatibility preservation condition
- Introduced success/failure automata for synchronous model
  - Encodes assumptions and guarantees
  - Composition computes assumption/guarantee propagation
  - Discussed compatibility for open systems
  - Asynchronous models can be handled similarly

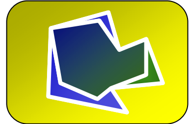
Model-independent formal framework



$proj(B)(\beta \,||\, q) \leq p'$

$\leftrightarrow$

$\beta \leq mirror(q \,||\, proj(B)(mirror(p')))$



$p \leq p' \leftrightarrow p \,||\, mirror(p') \in G$

$p \leq p'$ and $q \leq q' \rightarrow p \,||\, q \leq p' \,||\, q'$

p compatible with $q \leftrightarrow p \,||\, q \in G$

To probe further

- Interface models useful in simulation
  - Assumptions and guarantees can be turned into monitors that flag error conditions
- Interface models useful for synthesis
  - Assumptions provide don't care conditions for optimization
  - Synthesis problem formulated and solved using the *mirror* operator
  - Technique applicable to a variety of problems
- Interface models can be used for deployment
  - Platform gives performance assumptions and guarantees
  - Compatibility can be reformulated as correct deployment of a function on an architecture
- Algebraic models can be related by homomorphism
  - Translate between domains of computation
  - Verify asynchronous implementation vs. synchronous assumptions

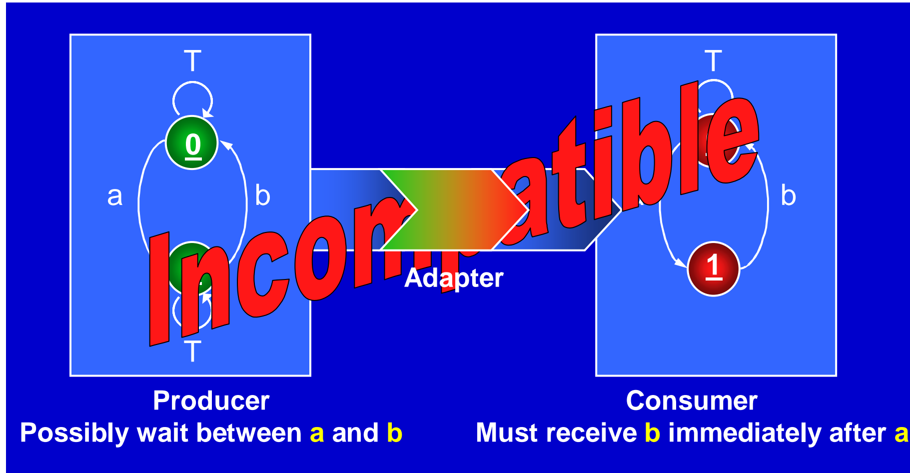
Bibliography

- Roberto Passerone. **Interface Specification and Converter Synthesis**. In Richard Zurawski, editor, *Embedded Systems Handbook*, Chapter 23. CRC Press, Taylor and Francis Group, Boca Raton, London, New York, 2005.
- Thomas A. Henzinger, Ranjit Jhala and Rupak Majumdar. **Permissive Interfaces**. *Proceedings of the Symposium on the Foundations of Software Engineering, 2005.* (FSE), 2005.
- Roberto Passerone, Luca de Alfaro, Thomas A. Henzinger, and Alberto Sangiovanni-Vincentelli. **Convertibility verification and converter synthesis: Two faces of the same coin**. *Proceedings of the International Conference on Computer-Aided Design* (ICCAD), IEEE Computer Society Press, 2002, pp. 132-139.
- Luca de Alfaro and Thomas A. Henzinger. **Interface theories for component-based design**. *Proceedings of the First International Workshop on Embedded Software* (EMSOFT), Lecture Notes in Computer Science 2211, Springer, 2001, pp. 148-165.
- Luca de Alfaro and Thomas A. Henzinger. **Interface automata**. *Proceedings of the Ninth Annual Symposium on Foundations of Software Engineering* (FSE), ACM Press, 2001, pp. 109-120.
- Radu Negulescu. **Process Spaces and the Formal Verification of Asynchronous Circuits**. PhD dissertation, University of Waterloo, Canada, 1998.
- Elizabeth S. Wolf. **Hierarchical Models of Synchronous Circuits for Formal Verification and Substitution**. PhD dissertation, Department of Computer Science, Stanford University, tech report number STAN-CS-TR-95-1557, October 1995.
- David L. Dill. **Trace Theory for Automatic Hierarchical Verification of Speed-Independent Circuits**. ACM Distinguished Dissertations, MIT Press, 1989.

Additional Slides

- Define problem of local specification synthesis as the synthesis of adapters

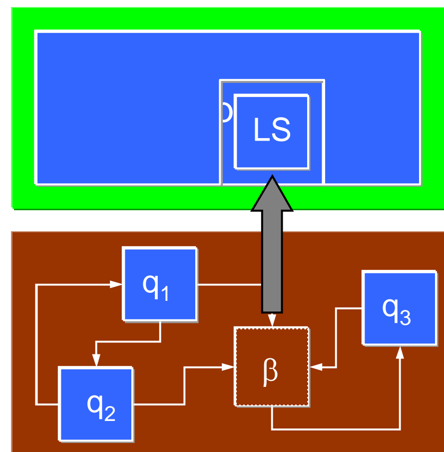
Example revisited



Data partitioned into two parts: a and b

Local specification synthesis

- Global specification *p'*
- Implementation
  - $p = \beta \;||\; q1 \;||\; q2 \;||\; q3$
- Find the local specification of a component β such that *p* refines *p'*
- The components around β form its context
- Solution



$\beta \leq \text{mirror}(q_1 \;||\; q_2 \;||\; q_3 \;||\; \text{proj}(A)(\text{mirror}(p')))$

Local specification synthesis

$$\text{proj}(B)(\beta \,||\, q) \leq p'$$

if and only if

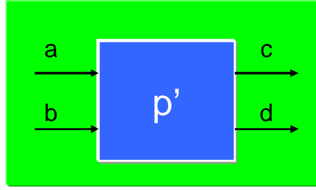
$$\beta \leq \text{mirror}(q \,||\, \text{proj}(B)(\text{mirror}(p')))$$

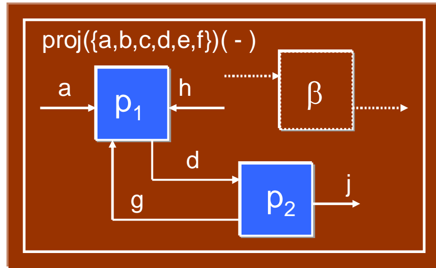
- Result similar to many specific results
  - But the formulation here is generic
  - In particular we don't say what mirror is
  - Complexity depends on the model and its implementation
  - Having established correctness, we can now focus on efficiency issues

Application areas

- Supervisory control
  - Context: a plant to be controlled
  - Global specification: what the plant is supposed to do
  - Local specification: controller for the plant
- Engineering Change (ECO), Rectification
  - Global specification: corrected functionality
  - Context: untouchable part of the system
  - Local specification: replacement part
- Optimization
  - Global specification: old functionality
  - Context: rest of the system
  - Local specification: optimized part
- Protocol conversion

Example

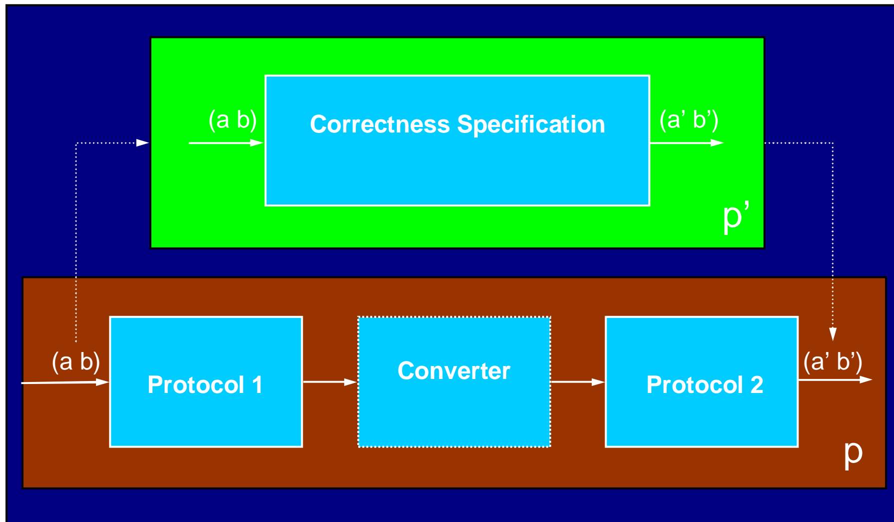




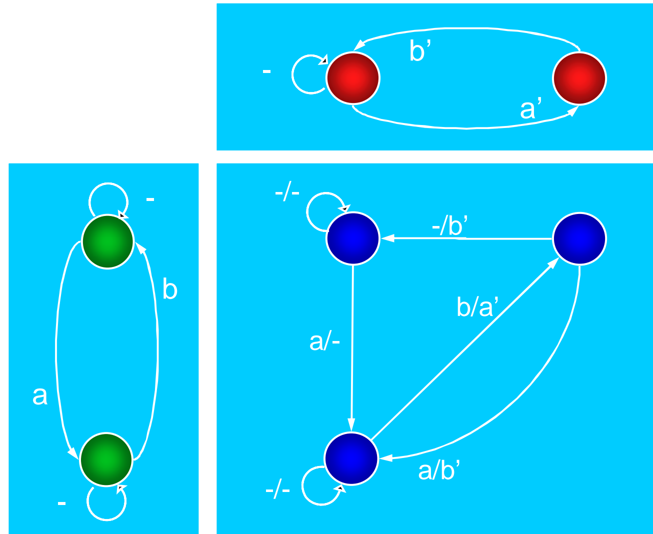
- $p' = (\{a, b\}, \{c, d\})$
- $p =$ proj(B)($\beta \,||\, p_1 \,||\, p_2$)
  - $p_1 = (\{ a, g, h \}, \{ d \})$
  - $p_2 = (\{ d \}, \{ g, j \})$
  - $B = \{ a, b, c, d, e, f \}$

- Solution: $\beta = (I, O)$ such that
- $I \cap O = \varnothing$
- $\varnothing \subseteq I \subseteq \{ a, b, d, g, j \}$
- $\{ c, h \} \subseteq O \subseteq \mathbf{A} - \{ a, b, d, e, f, g, j \}$

Protocol Conversion



Conversion result



Additional slides

- Define an asynchronous model, and relate it to the synchronous one

Asynchronous Agent Algebra

- Let A = { a, b, c } be a set of signals
  - Behaviors are non-empty sequences of signals $\mathcal{L}_A = (2^A - \varnothing)^*$
  - x = < {a}, {b}, {bc}, {ab}, {b}, {ac}, … >
  - $p = (A, P)$ where $P \subseteq (2^A - \varnothing)^*$
- Parallel composition operator
  - $p_1 || p_2 = (A_1 \cup A_2, P_1 \cap P_2)$
- Refinement order
  - $p_1 \leq p_2$ if and only if $P_1 \subseteq P_2$
- Mirror
  - mirror(p) = (A, $\mathcal{L}_A - P$)

Desynchronization

- Abstraction
  - x' = < {a}, {b}, {bc}, {ab}, {b}, {ac}, … > is an *abstraction* of
  - x_1 = < {a}, {b}, ∅, {bc}, {ab}, ∅, {b}, {ac}, … >,
  - x_2 = < {a}, ∅, {b}, {bc}, ∅, {ab}, ∅, ∅, {b}, {ac}, … >, etc.
  - x_1 and x_2 are *concretizations* of x'
- Upper Bound
  - $\Psi_u(p)$ contains the abstraction of *all* the behaviors of p
  - $\Psi_u(p)$ contains the abstraction of behaviors that are not necessarily in p
  - The abstraction removes the ability to *count* the clocks
- Lower Bound
  - $\Psi_l(p)$ contains the abstraction of *only* behaviors of p
  - If x' is a behavior of $\Psi_l(p)$, then all the concretizations of x' must be in p
  - $\Psi_l(p)$ contains the behaviors of p that are insensitive to delays
- Ψ_l and Ψ_u form a conservative approximation

Desynchronization

- Assume *p* is such that $\Psi_u(p) = \Psi_l(p) = p'$
  - *p'* has *all* the behaviors of *p*
  - *p'* has *only* the behaviors of *p*
  - All the behaviors of *p* are insensitive to delays
- Therefore *p* is a synchronous agent with non-deterministic delay
  - Hence it can be represented exactly in the asynchronous model
- All asynchronous agents have a corresponding non-deterministic synchronous agent
  - The inverse of the conservative approximation is defined everywhere

Additional slides

- Computing mirrors when they don't exist

Mirrors don't always exist

- Mirrors may not exist for two reasons
  - Parallel composition is unable to characterize the refinement relation
  - The model fails to have maximally compatible agents
- An extended model is required
  - Agents that contain more information
  - The original model is embedded in the extended model
  - The larger model improves our understanding of the original (much like real and complex numbers)

Extension of I/O agents

- Locked I/O agents
  - $p = (I, O, L)$
  - $p_1 \| p_2 = ((I_1 \cup I_2) - (O_1 \cup O_2), O_1 \cup O_2, L_1 \cup L_2)$ iff $(O_1 \cup L_1) \cap (O_2 \cup L_2) = \varnothing$, $I_1 \cap L_2 = \varnothing$, $I_2 \cap L_1 = \varnothing$
  - mirror(p) $= (O, I, \mathcal{A} - (I \cup O \cup L))$
  - Embedding : $(I, O) \rightarrow (I, O, \varnothing)$