EECS 144/244: Fundamental Algorithms for System Modeling, Analysis, and Optimization

Discrete Systems

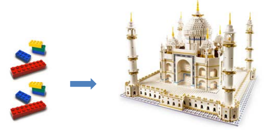
Lecture: Contracts, Asynchronous Composition, Fairness

Stavros Tripakis

University of California, Berkeley

ASSUMPTIONS, GUARANTEES, CONTRACTS

Component-Based Design



Composition: what for?

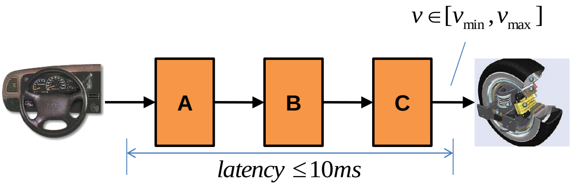
- Building large systems from smaller components (subsystems).

Important and related notions and questions:

- Modularity: what are the right components? how independent are they from each other?
- Reusability: what are the right components? how generic/reusable are they?
- Compatibility/Composability: can two components be composed?
- Compositionality: many meanings, e.g., can the properties of the overall system be derived from those of its subsystems?
- Substitutability: when can a component replace another one?
- Incrementality: can a component be added "later"?
- Reconfigurability
- ...

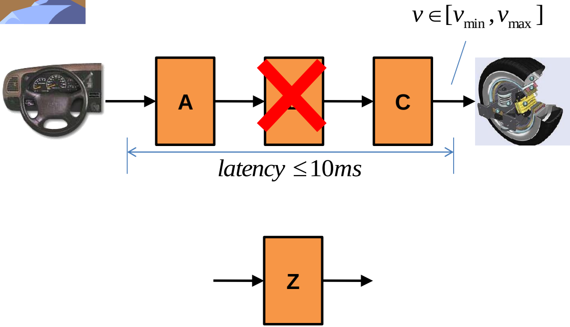
Overview: contracts as behavioral types

Substitutability



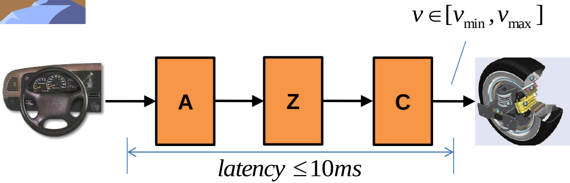
1

Substitutability

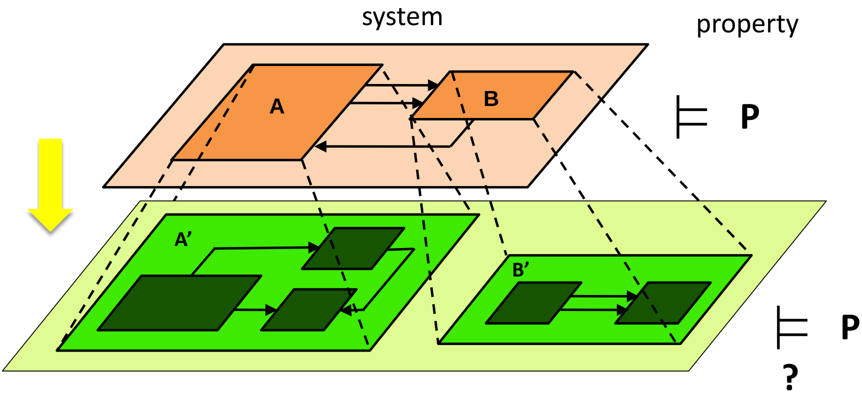


Substitutability

How to ensure properties are preserved?



System refinement



4

Interface theories [Alfaro, Henzinger, et al.]

- Interface = component abstraction
- Interface composition: A • B = C
  - Check compatibility here! (local, lightweight)
- Interface refinement: A' ≤ A
- Theorems:

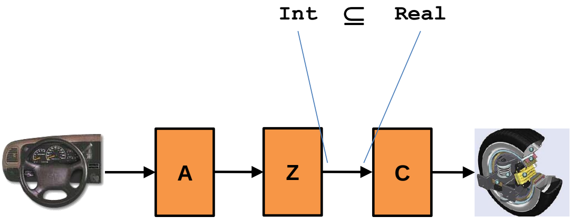
(1) If A' ≤ A and A satisfies P then A' satisfies P.

(2) If A' ≤ A and B' ≤ B, then A' • B' ≤ A • B.

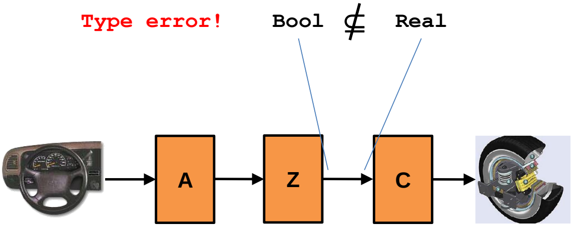
(1) and (2) => substitutability

5

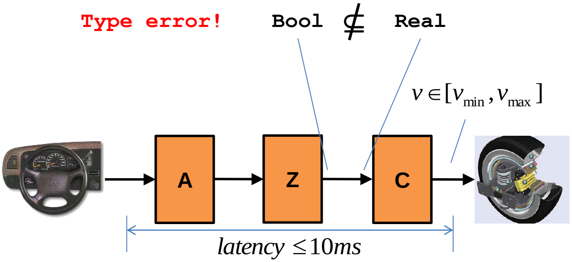
Type theories



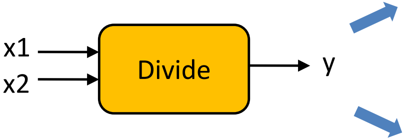
Type theories



Interface theories = **behavioral** type theories



Relational interfaces



block

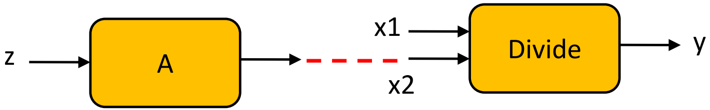
`(Real, Real) -> Real`

standard type

$$\boxed{x_2 \neq 0 \wedge \left(x_1 > 0 \wedge x_2 > 0 \Rightarrow y > 0\right)}$$

relational interface

Relational interfaces: type checking

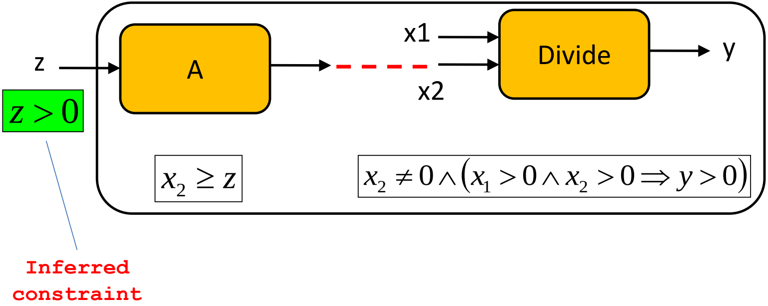


$$\boxed{x_2 \geq 0} \quad \not\Rightarrow \quad \boxed{x_2 \neq 0 \wedge \left(x_1 > 0 \wedge x_2 > 0 \Rightarrow y > 0\right)}$$

Type error!

12

Relational interfaces: type inference



13

Subtyping for substitutability

$$\phi' \prec \phi \quad \stackrel{\text{def}}{=} \quad \boxed{\begin{array}{l} in(\phi) \Rightarrow in(\phi') \\ (in(\phi) \wedge \phi') \Rightarrow \phi \end{array}}$$

ϕ' subtype of ϕ

$\Rightarrow$ (and sometimes $\Leftrightarrow$)

ϕ' can replace ϕ in any context

Can be computed using SAT/SMT solvers

Key aspects of contracts

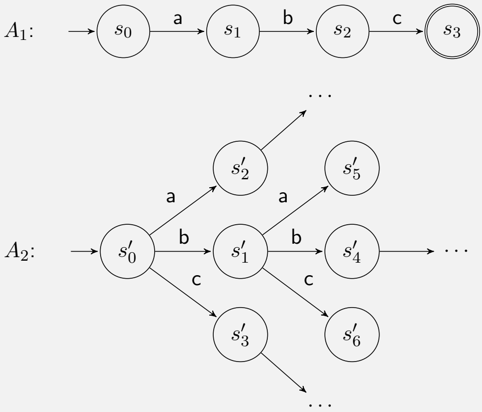
Inputs, outputs.

Assumptions vs. requirements on inputs.

Guarantees on outputs.

Recall: total vs. partial transition functions

Suppose $\Sigma = a, b, c$.



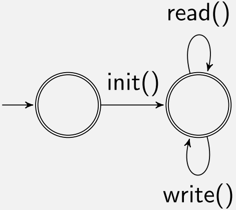
What if the transition function of the "receiver" A_2 is also partial?

Recall: non-input-completeness

Different meanings and usages of partial inputs:

- **Requirements**: I require that the environment never provides this input (at that time).
  - This can be useful for *contract-based design*.
  - More about this when we talk about composition.

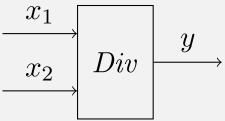
  Example:

- Assumptions: I know that the environment will never provide this input (at that time).

Assumptions vs. Requirements on the Inputs

Example: Division component.



Two possible ways to look at its contract:

- Assumption on inputs:

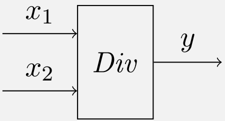
$$x_2 \neq 0 \rightarrow y = \frac{x_1}{x_2}$$

- Requirements on inputs:

$$x_2 \neq 0 \wedge y = \frac{x_1}{x_2}$$

Assumptions vs. Requirements on the Inputs

Example: Division component.



Two possible ways to look at its contract:

- Assumption on inputs:

$$x_2 \neq 0 \rightarrow y = \frac{x_1}{x_2}$$

- Requirements on inputs:

$$x_2 \neq 0 \wedge y = \frac{x_1}{x_2}$$

As we shall see, these have different implications during composition.

Formalizing Contracts

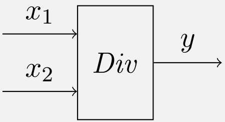
Contracts for synchronous components: *relational interfaces* [Tripakis et al., 2011].

Generalizations of Mealy machines:

- Finite sets of input and output variables.
- Set of nodes (like the states of a Mealy machine).
- Every node annotated by a predicate on input and output variables: *static contract* (holds at a given step).
- Transitions between nodes specify *dynamic* contracts.

Relational Interfaces: the Stateless Case

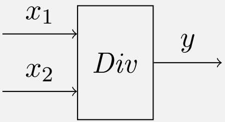
Example: Division interface.



$$Div \;=\; (\underbrace{\{x_1, x_2\}}_{\text{input variables}}, \underbrace{\{y\}}_{\text{output variables}}, \underbrace{x_2 \neq 0 \wedge y = \frac{x_1}{x_2}}_{\text{“static” contract}})$$

Relational Interfaces: the Stateless Case

Example: Division interface.



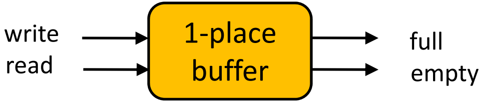
$$Div \quad = \quad (\underbrace{\{x_1, x_2\}}_{\text{input variables}} , \quad \underbrace{\{y\}}_{\text{output variables}} , \underbrace{x_2 \neq 0 \wedge y = \frac{x_1}{x_2}}_{\text{“static” contract}})$$

Meaning: at every synchronous step:

1. Environment proposes inputs x_1, x_2.
   - If x_1, x_2 already violate the contract (if $x_2 = 0$), environment is to blame. Otherwise:
2. Component chooses output y.
   - If x_1, x_2, y violate the contract, component is to blame.

Relational Interfaces: the Stateful Case

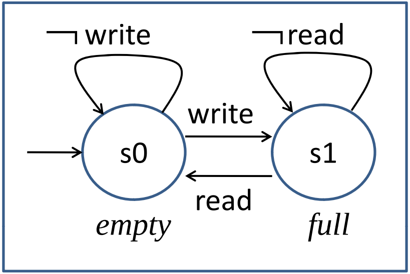
Example: single-place buffer.



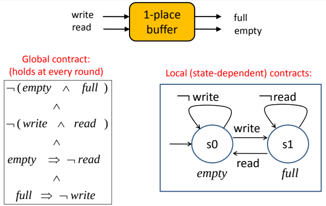
Global contract:
(holds at every round)

$$\neg(\mathit{empty} \wedge \mathit{full}) \\ \wedge \\ \neg(\mathit{write} \wedge \mathit{read}) \\ \wedge \\ \mathit{empty} \Rightarrow \neg\mathit{read} \\ \wedge \\ \mathit{full} \Rightarrow \neg\mathit{write}$$

Local (state-dependent) contracts:



Relational Interfaces: the Stateful Case

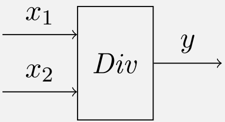


Meaning: at every synchronous step:

0. Contract := contract of current state.
1. Environment proposes inputs x_1, x_2.
   - If x_1, x_2 already violate the contract (if $x_2 = 0$), environment is to blame. Otherwise:
2. Component chooses output y.
   - If x_1, x_2, y violate the contract, component is to blame.
3. Find which guard of the automaton is satisfied by the vector x_1, x_2, y (guard must be unique $\Rightarrow$ determinism).
4. Take corresponding transition, updating automaton state (and therefore also the contract that must hold on the *next* step).

Looking more closely at our contracts

Division interface with non-deterministic output.



$$
\begin{aligned}
Div &\ \widehat{=}\ \ (\{x_1, x_2\}, \{y\}, \phi_{Div}) \\
\phi_{Div} &\ \widehat{=}\ \ x_2 \neq 0 \wedge \phi_{sign} \\
\phi_{sign} &\ \widehat{=}\ \ (y = 0 \leftrightarrow x_1 = 0) \wedge \big(y < 0 \leftrightarrow (x_1 < 0 < x_2 \vee x_2 < 0 < x_1)\big)
\end{aligned}
$$

If $x_1 = x_2 = 1$, output can be any $y > 0$.

- Very useful for abstraction.

Looking more closely at our contracts

$$
\begin{aligned}
Div &\mathrel{\widehat{=}} (\{x_1, x_2\}, \{y\}, \phi_{Div}) \\
\phi_{Div} &\mathrel{\widehat{=}} x_2 \neq 0 \wedge \phi_{sign} \\
\phi_{sign} &\mathrel{\widehat{=}} (y = 0 \leftrightarrow x_1 = 0) \wedge \big(y < 0 \leftrightarrow (x_1 < 0 < x_2 \vee x_2 < 0 < x_1)\big)
\end{aligned}
$$

- **Relational**: e.g., $y = 0 \leftrightarrow x_1 = 0$
  - as opposed to simple types like: `Real` $\times$ `Real` $\rightarrow$ `Real`

Looking more closely at our contracts

$$
\begin{array}{rcl}
Div & \widehat{=} & (\{x_1, x_2\}, \{y\}, \phi_{Div}) \\
\phi_{Div} & \widehat{=} & x_2 \neq 0 \wedge \phi_{sign} \\
\phi_{sign} & \widehat{=} & (y = 0 \leftrightarrow x_1 = 0) \wedge \big(y < 0 \leftrightarrow (x_1 < 0 < x_2 \vee x_2 < 0 < x_1)\big)
\end{array}
$$

- **Relational**: e.g., $y = 0 \leftrightarrow x_1 = 0$
  - as opposed to simple types like: `Real` $\times$ `Real` $\rightarrow$ `Real`
- **Non-deterministic**: e.g., if $x_1 = x_2 = 1$, output can be any $y > 0$
  - very useful for abstraction

Looking more closely at our contracts

$$
\begin{aligned}
Div &\mathrel{\widehat{=}} (\{x_1, x_2\}, \{y\}, \phi_{Div}) \\
\phi_{Div} &\mathrel{\widehat{=}} x_2 \neq 0 \land \phi_{sign} \\
\phi_{sign} &\mathrel{\widehat{=}} (y = 0 \leftrightarrow x_1 = 0) \land \big(y < 0 \leftrightarrow (x_1 < 0 < x_2 \lor x_2 < 0 < x_1)\big)
\end{aligned}
$$

- **Relational**: e.g., $y = 0 \leftrightarrow x_1 = 0$
  - as opposed to simple types like: `Real` $\times$ `Real` $\rightarrow$ `Real`
- **Non-deterministic**: e.g., if $x_1 = x_2 = 1$, output can be any $y > 0$
  - very useful for abstraction
- **Non-input-complete**: $x_2 \neq 0 \land \phi_{sign}$
  - as opposed to: $x_2 \neq 0 \rightarrow \phi_{sign}$

Looking more closely at our contracts

$$
\begin{array}{rcl}
Div & \widehat{=} & (\{x_1, x_2\}, \{y\}, \phi_{Div}) \\
\phi_{Div} & \widehat{=} & x_2 \neq 0 \wedge \phi_{sign} \\
\phi_{sign} & \widehat{=} & (y = 0 \leftrightarrow x_1 = 0) \wedge \big(y < 0 \leftrightarrow (x_1 < 0 < x_2 \vee x_2 < 0 < x_1)\big)
\end{array}
$$

- **Relational**: e.g., $y = 0 \leftrightarrow x_1 = 0$
  - as opposed to simple types like: `Real` $\times$ `Real` $\rightarrow$ `Real`
- **Non-deterministic**: e.g., if $x_1 = x_2 = 1$, output can be any $y > 0$
  - very useful for abstraction
- **Non-input-complete**: $x_2 \neq 0 \wedge \phi_{sign}$
  - as opposed to: $x_2 \neq 0 \rightarrow \phi_{sign}$
  - this will make things a bit complicated ...
  - do we really need it?

Why non-input-complete contracts are useful

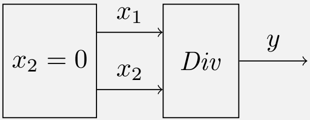
Consider the alternative contract

$$x_2 \neq 0 \rightarrow \phi_{sign}$$

- This allows y to take any value when $x_2 = 0$.
- But it also assumes that y will take *some* value!
- What if the component "breaks" when fed with illegal inputs?
  - e.g., algorithm may not terminate when inputs are illegal
  - hardware may "burn up" when input voltage is too high.

Why non-input-complete contracts are useful (continued)

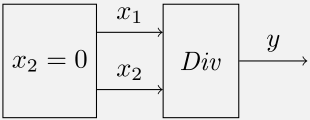
Catching incompatible compositions early:



If the contract of Div is $x_2 \neq 0 \wedge \cdots$ then the composite contract is $false$, indicating **incompatibility**.

Why non-input-complete contracts are useful (continued)

Catching incompatible compositions early:

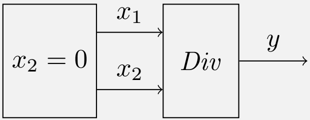


If the contract of Div is $x_2 \neq 0 \wedge \cdots$ then the composite contract is $false$, indicating **incompatibility**.

If the contract of Div is $x_2 \neq 0 \rightarrow \cdots$ then the composite contract (after hiding x_2) is $true$. How to interpret this?

Why non-input-complete contracts are useful (continued)

Catching incompatible compositions early:



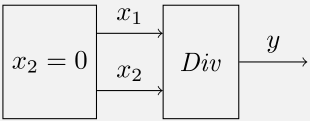
If the contract of Div is $x_2 \neq 0 \wedge \cdots$ then the composite contract is $false$, indicating **incompatibility**.

If the contract of Div is $x_2 \neq 0 \rightarrow \cdots$ then the composite contract (after hiding x_2) is $true$. How to interpret this?

- We cannot interpret it as "incompatible": $true$ may simply mean "nothing is known about this component".

Why non-input-complete contracts are useful (continued)

Catching incompatible compositions early:



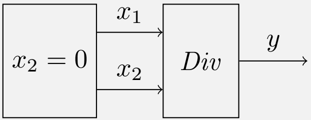
If the contract of Div is $x_2 \neq 0 \wedge \cdots$ then the composite contract is $false$, indicating **incompatibility**.

If the contract of Div is $x_2 \neq 0 \rightarrow \cdots$ then the composite contract (after hiding x_2) is $true$. How to interpret this?

- We cannot interpret it as "incompatible": $true$ may simply mean "nothing is known about this component".
- We could try to verify the composition against a specific property, e.g., $y \in [L, U]$.
  - Not easy to come up with such properties.
  - May not want to do "full" verification.

Why non-input-complete contracts are useful (continued)

Catching incompatible compositions early:



If the contract of Div is $x_2 \neq 0 \wedge \cdots$ then the composite contract is $false$, indicating **incompatibility**.

If the contract of Div is $x_2 \neq 0 \rightarrow \cdots$ then the composite contract (after hiding x_2) is $true$. How to interpret this?

- We cannot interpret it as "incompatible": $true$ may simply mean "nothing is known about this component".
- We could try to verify the composition against a specific property, e.g., $y \in [L, U]$.
  - Not easy to come up with such properties.
  - May not want to do "full" verification.
  - Instead: "light-weight" **type-checking**.

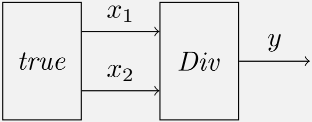
So ...

- relational + non-deterministic + non-input-complete contracts are good.

So ...

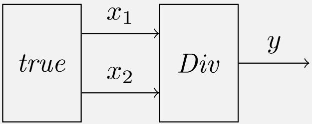
- relational + non-deterministic + non-input-complete contracts are good.
- Next: from such contracts, (non-standard) definitions of composition and refinement appear to follow inevitably.

Serial composition



How should we define the composite contract?

Serial composition



How should we define the composite contract?

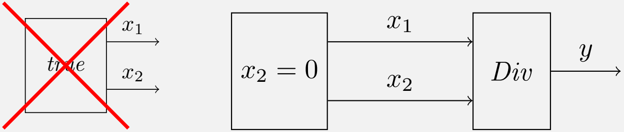
- Standard definition: composition = conjunction

$$true \wedge x_2 \neq 0 \wedge \cdots$$

- this does not seem to indicate any incompatibility ...

Serial composition: problem with standard definition

What if we replace $true$ with $x_2 = 0$?



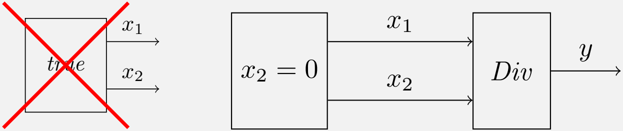
- Standard definition: composition = conjunction

$$x_2 = 0 \wedge x_2 \neq 0 \wedge \cdots \quad \equiv \quad false$$

- this indicates incompatibility ...

Serial composition: problem with standard definition

What if we replace $true$ with $x_2 = 0$?



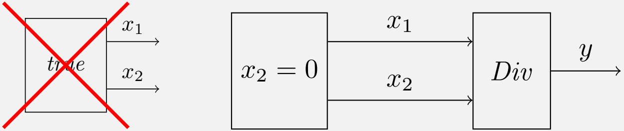
- Standard definition: composition = conjunction

$$x_2 = 0 \wedge x_2 \neq 0 \wedge \cdots \quad \equiv \quad \mathit{false}$$

- this indicates incompatibility ...
- Yet $x_2 = 0$ seems a valid substitute for $true$: it more deterministic, i.e., "more defined". It should be a valid *refinement* of $true$.

Serial composition: problem with standard definition

What if we replace $true$ with $x_2 = 0$?

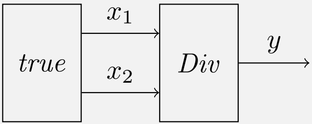


- Standard definition: composition = conjunction

$$x_2 = 0 \wedge x_2 \neq 0 \wedge \cdots \quad \equiv \quad \mathit{false}$$

- this indicates incompatibility ...
- Yet $x_2 = 0$ seems a valid substitute for $true$: it more deterministic, i.e., "more defined". It should be a valid *refinement* of $true$.
- Conclusion: The standard definition violates preservation of refinement by composition ... ☹

Serial composition: alternative definition

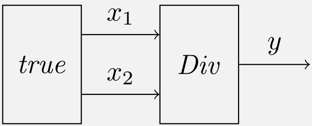


Instead, we define the composite contract as follows:

- "**Demonic**" non-determinism: 

$$true \wedge x_2 \neq 0 \wedge \cdots \wedge \underbrace{(\forall x_2 : true \rightarrow x_2 \neq 0)}_{\text{this is the additional constraint}}$$

Serial composition: alternative definition



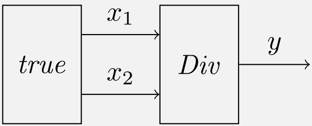
Instead, we define the composite contract as follows:

- "**Demonic**" non-determinism: 

$$
\mathit{true} \land x_2 \neq 0 \land \cdots \land \underbrace{(\forall x_2 : \mathit{true} \to x_2 \neq 0)}_{\text{this is the additional constraint}}
$$

$$
\forall x_2 : \mathit{true} \to x_2 \neq 0 \quad \equiv \quad \forall x_2 : x_2 \neq 0 \quad \equiv \quad \mathit{false}
$$

Serial composition: alternative definition



Instead, we define the composite contract as follows:

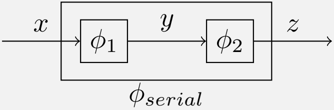
- "**Demonic**" non-determinism: 

$$true \wedge x_2 \neq 0 \wedge \cdots \wedge \underbrace{(\forall x_2 : true \rightarrow x_2 \neq 0)}_{\text{this is the additional constraint}}$$

$$\forall x_2 : true \rightarrow x_2 \neq 0 \quad \equiv \quad \forall x_2 : x_2 \neq 0 \quad \equiv \quad false$$

- Incompatibility detected!

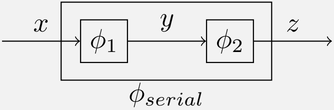
Serial composition: general case



- Composite contract:

$$\phi_{serial} \quad \widehat{=} \quad \phi_1 \wedge \phi_2 \wedge (\forall y : \phi_1 \rightarrow \exists z : \phi_2)$$

Serial composition: general case



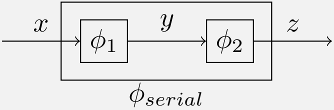
- Composite contract:

$$\phi_{serial} \quad \widehat{=} \quad \phi_1 \wedge \phi_2 \wedge (\forall y : \phi_1 \rightarrow \exists z : \phi_2)$$

- Let $\mathsf{in}(\phi_2) \;\widehat{=}\; \exists z : \phi_2$. Then

$$\phi_{serial} \quad \widehat{=} \quad \phi_1 \wedge \phi_2 \wedge \big(\forall y : \phi_1 \rightarrow \mathsf{in}(\phi_2)\big)$$

Serial composition: general case



- Composite contract:

$$\phi_{serial} \quad \widehat{=} \quad \phi_1 \wedge \phi_2 \wedge (\forall y : \phi_1 \rightarrow \exists z : \phi_2)$$

- Let $\mathsf{in}(\phi_2) \ \widehat{=}\ \exists z : \phi_2$. Then

$$\phi_{serial} \quad \widehat{=} \quad \phi_1 \wedge \phi_2 \wedge \big(\forall y : \phi_1 \rightarrow \mathsf{in}(\phi_2)\big)$$

- Note: if ϕ_2 is input-complete or ϕ_1 is deterministic, then $\phi_{serial} \equiv \phi_1 \wedge \phi_2$.

Refinement

$$\xrightarrow{x} \boxed{\phi_2} \xrightarrow{y} \quad \sqsubseteq \quad \xrightarrow{x} \boxed{\phi_1} \xrightarrow{y}$$

When can we say that ϕ_2 is a valid refinement of ϕ_1?

Refinement

$$\xrightarrow{x} \boxed{\phi_2} \xrightarrow{y} \qquad \sqsubseteq \qquad \xrightarrow{x} \boxed{\phi_1} \xrightarrow{y}$$

When can we say that ϕ_2 is a valid refinement of ϕ_1?

- Standard view: refinement = implication

$$\phi_2 \sqsubseteq \phi_1 \quad \widehat{=} \quad \phi_2 \rightarrow \phi_1$$

Refinement

$$\xrightarrow{x} \boxed{\phi_2} \xrightarrow{y} \qquad \sqsubseteq \qquad \xrightarrow{x} \boxed{\phi_1} \xrightarrow{y}$$

When can we say that ϕ_2 is a valid refinement of ϕ_1?

- Standard view: refinement = implication

$$\phi_2 \sqsubseteq \phi_1 \quad \widehat{=} \quad \phi_2 \to \phi_1$$

- Does not work (refinement does not preserve compatibility):

$\boxed{x = 0} \xrightarrow{x} \boxed{\mathit{true}}$ compatible

$\boxed{x = 0} \xrightarrow{x} \boxed{x \neq 0}$ incompatible, even though $x \neq 0 \to \mathit{true}$

Refinement

$$\xrightarrow{x} \boxed{\phi_2} \xrightarrow{y} \qquad \sqsubseteq \qquad \xrightarrow{x} \boxed{\phi_1} \xrightarrow{y}$$

When can we say that ϕ_2 is a valid refinement of ϕ_1?

- Standard view: refinement = implication

$$\phi_2 \sqsubseteq \phi_1 \quad \widehat{=} \quad \phi_2 \to \phi_1$$

- Does not work (refinement does not preserve compatibility):

$\boxed{x = 0} \xrightarrow{x} \boxed{\mathit{true}}$ compatible

$\boxed{x = 0} \xrightarrow{x} \boxed{x \neq 0}$ incompatible, even though $x \neq 0 \to \mathit{true}$

Need to treat inputs and outputs differently.

Alternating Refinement

$$\xrightarrow{x} \boxed{\phi_2} \xrightarrow{y} \qquad \sqsubseteq \qquad \xrightarrow{x} \boxed{\phi_1} \xrightarrow{y}$$

- Alternating refinement:

$$\phi_2 \sqsubseteq \phi_1 \quad \widehat{=} \quad \big(\mathsf{in}(\phi_1) \to \mathsf{in}(\phi_2)\big) \wedge \Big(\big(\mathsf{in}(\phi_1) \wedge \phi_2\big) \to \phi_1\Big)$$

Alternating Refinement

- Alternating refinement:

$$\phi_2 \sqsubseteq \phi_1 \quad \widehat{=} \quad \big(\mathsf{in}(\phi_1) \to \mathsf{in}(\phi_2)\big) \land \Big(\big(\mathsf{in}(\phi_1) \land \phi_2\big) \to \phi_1\Big)$$

If ϕ_1, ϕ_2 are input-complete, then $(\phi_2 \sqsubseteq \phi_1) \;\equiv\; (\phi_2 \to \phi_1)$.

Alternating Refinement

- Alternating refinement:

$$\phi_2 \sqsubseteq \phi_1 \quad \widehat{=} \quad \big(\mathsf{in}(\phi_1) \to \mathsf{in}(\phi_2)\big) \land \Big(\big(\mathsf{in}(\phi_1) \land \phi_2\big) \to \phi_1\Big)$$

If ϕ_1, ϕ_2 are input-complete, then $(\phi_2 \sqsubseteq \phi_1) \;\equiv\; (\phi_2 \to \phi_1)$.

Main result [Tripakis et al., 2011]:

- Refinement is equivalent to substitutability (*fully abstract*): $\phi_2 \sqsubseteq \phi_1$ *iff* ϕ_2 *can replace* ϕ_1 *in any context*.

Alternating Refinement

$$\xrightarrow{x}\boxed{\phi_2}\xrightarrow{y} \qquad \sqsubseteq \qquad \xrightarrow{x}\boxed{\phi_1}\xrightarrow{y}$$

- Alternating refinement:

$$\phi_2 \sqsubseteq \phi_1 \quad \widehat{=} \quad \big(\mathsf{in}(\phi_1) \to \mathsf{in}(\phi_2)\big) \land \Big(\big(\mathsf{in}(\phi_1) \land \phi_2\big) \to \phi_1\Big)$$

  If ϕ_1, ϕ_2 are input-complete, then $(\phi_2 \sqsubseteq \phi_1) \equiv (\phi_2 \to \phi_1)$.

Main result [Tripakis et al., 2011]:

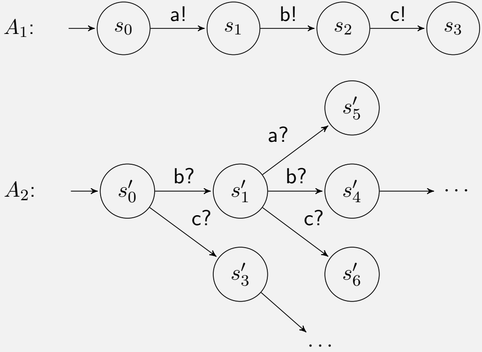
- Refinement is equivalent to substitutability (*fully abstract*): $\phi_2 \sqsubseteq \phi_1$ *iff* ϕ_2 *can replace* ϕ_1 *in any context*.
- Note that other definitions are sufficient but not necessary for substitutability, e.g.:

$$\big(\mathsf{in}(\phi_1) \to \mathsf{in}(\phi_2)\big) \land \big(\phi_2 \to \phi_1\big)$$

  C.f. Liskov-Wing's behavioral subtyping [Liskov and Wing, 1994].

Contracts in the Automata World

Interface Automata [de Alfaro and Henzinger, 2001].



The composition of A_1 and A_2 is invalid: A_1 offers a as output, but A_2 is not able to accept it as input.

ASYNCHRONOUS COMPOSITION

Recall: Synchronous Composition of Kripke Structures

Given two KS K_1 and K_2 with

$$K_i = (P_i, S_i, S_0^i, L_i, R_i)$$

the synchronous composition of K_1 and K_2 is a new KS

$$K_1 \times K_2 = (P_1 \cup P_2, S_1 \times S_2, S_0^1 \times S_0^2, L, R)$$

where

- $L\big((s_1, s_2)\big) = L_1(s_1) \cup L_2(s_2)$
- $\big((s_1, s_2), (s_1', s_2')\big) \in R$ iff $(s_1, s_1') \in R_1$ **and** $(s_2, s_2') \in R_2$

Asynchronous Composition of Kripke Structures

Given two KS K_1 and K_2 with

$$K_i = (P_i, S_i, S_0^i, L_i, R_i)$$

the asynchronous composition of K_1 and K_2 is a new KS

$$K_1 || K_2 = (P_1 \cup P_2, S_1 \times S_2, S_0^1 \times S_0^2, L, R)$$

where

- $L\big((s_1, s_2)\big) = L_1(s_1) \cup L_2(s_2)$
- $\big((s_1, s_2), (s_1', s_2')\big) \in R$ iff $(s_1, s_1') \in R_1$ and $s_2' = s_2$ **or** $(s_2, s_2') \in R_2$ and $s_1' = s_1$

Recall: Symbolic Synchronous Composition of Kripke Structures

Given two KS K_1 and K_2, each represented symbolically as

$$K_i = (Init_i, Trans_i)$$

their synchronous composition $K_1 \times K_2$ can be represented symbolically as

$$K_1 \times K_2 \quad = \quad (Init_1 \wedge Init_2, Trans_1 \wedge Trans_2)$$

Recall: Symbolic Synchronous Composition of Kripke Structures

Given two KS K_1 and K_2, each represented symbolically as

$$K_i = (Init_i, Trans_i)$$

their synchronous composition $K_1 \times K_2$ can be represented symbolically as

$$K_1 \times K_2 \quad = \quad (Init_1 \wedge Init_2, Trans_1 \wedge Trans_2)$$

How can the asynchronous composition $K_1 || K_2$ be represented symbolically?

Recall: Symbolic Synchronous Composition of Kripke Structures

Given two KS K_1 and K_2, each represented symbolically as

$$K_i = (Init_i, Trans_i)$$

their synchronous composition $K_1 \times K_2$ can be represented symbolically as

$$K_1 \times K_2 \quad = \quad (Init_1 \wedge Init_2, Trans_1 \wedge Trans_2)$$

How can the asynchronous composition $K_1 || K_2$ be represented symbolically?

$$K_1 || K_2 \quad = \quad (Init_1 \wedge Init_2, Trans_1 \vee Trans_2)$$

is this correct?

Symbolic Asynchronous Composition of Kripke Structures

Given two KS K_1 and K_2, each represented symbolically as

$$K_i = (Init_i, Trans_i)$$

their asynchronous composition $K_1||K_2$ can be represented symbolically as

$$K_1||K_2 \quad = \quad \big(Init_1 \wedge Init_2, (Trans_1 \wedge \vec{x}_2' = \vec{x}_2) \vee (Trans_2 \wedge \vec{x}_1' = \vec{x}_1)\big)$$

where $\vec{x_i}$ are the variables of K_i.

Symbolic Asynchronous Composition of Kripke Structures

Given two KS K_1 and K_2, each represented symbolically as

$$K_i = (Init_i, Trans_i)$$

their asynchronous composition $K_1||K_2$ can be represented symbolically as

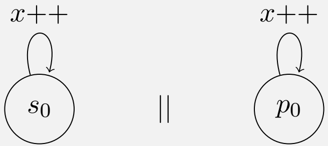
$$K_1||K_2 \quad = \quad \big(Init_1 \wedge Init_2, (Trans_1 \wedge \vec{x}_2' = \vec{x}_2) \vee (Trans_2 \wedge \vec{x}_1' = \vec{x}_1)\big)$$

where $\vec{x_i}$ are the variables of K_i.

Is it correct now?

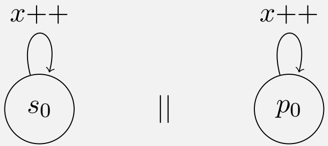
Symbolic Asynchronous Composition of Kripke Structures

Consider two asynchronous processes writing to a shared variable x:



Symbolic Asynchronous Composition of Kripke Structures

Consider two asynchronous processes writing to a shared variable x:

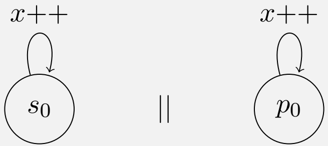


Composite transition relation:

$$\underbrace{x' = x + 1 \land x' = x}_{\text{from process 1}} \lor \underbrace{x' = x + 1 \land x' = x}_{\text{from process 2}}$$

Symbolic Asynchronous Composition of Kripke Structures

Consider two asynchronous processes writing to a shared variable x:

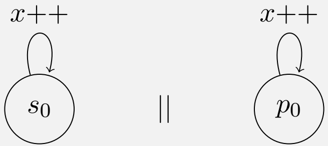


Composite transition relation:

$$\underbrace{x' = x + 1 \wedge x' = x}_{\text{from process 1}} \vee \underbrace{x' = x + 1 \wedge x' = x}_{\text{from process 2}} \quad \equiv \quad \mathit{false}$$

Symbolic Asynchronous Composition of Kripke Structures

Consider two asynchronous processes writing to a shared variable x:



Composite transition relation:

$$\underbrace{x' = x + 1 \land x' = x}_{\text{from process 1}} \lor \underbrace{x' = x + 1 \land x' = x}_{\text{from process 2}} \quad \equiv \quad \mathit{false}$$

Need to talk explicitly about shared variables.

Symbolic Asynchronous Composition of Kripke Structures

Given two KS K_1 and K_2, each represented symbolically as

$$K_i = (Init_i, Trans_i)$$

their asynchronous composition $K_1||K_2$ can be represented symbolically as

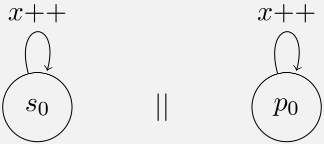
$$K_1||K_2 \quad = \quad \big(Init_1 \wedge Init_2, (Trans_1 \wedge \vec{x}_2' = \vec{x}_2) \vee (Trans_2 \wedge \vec{x}_1' = \vec{x}_1)\big)$$

where:

- $\vec{x_i}$ are the variables *owned by* K_i: they are written *only by* K_i (they can be read by K_j, $j \neq i$).
- $Trans_i$ may refer also to *shared variables* $\vec{v}$, which are written by both K_1 and K_2.

Symbolic Asynchronous Composition of Kripke Structures

Consider two asynchronous processes writing to a shared variable x:



Composite transition relation:

$$\underbrace{x' = x + 1}_{\text{from process 1}} \vee \underbrace{x' = x + 1}_{\text{from process 2}}$$

Only one variable, x, shared.

Asynchronous Process Communication

Two prominent paradigms:

- Shared memory
  - A common pool of shared (global) variables
  - Common problems: avoid corrupt values, races, deadlocks (e.g., when semaphores are used), ...
- Message passing
  - Generally "cleaner" (but perhaps more difficult to implement)
  - Can even ensure determinism in some cases! (see Kahn Process Networks later in this course)

Spin / Promela offers both [Holzmann, 2003].

NuSMV offers synchronous composition (asynchronous is deprecated).

DIGRESSION:

FORMALISMS, LANGUAGES and TOOLS

Languages vs. Formalisms

Formalisms: abstract, mathematical objects.

Languages implement formalisms: they have concrete syntax.
They usually come together with *tools*.

Example:

- Formalism: FSM.
- Language: the language of NuSMV.

Languages vs. Formalisms

Formalisms: abstract, mathematical objects.

Languages implement formalisms: they have concrete syntax. They usually come together with *tools*.

Example:

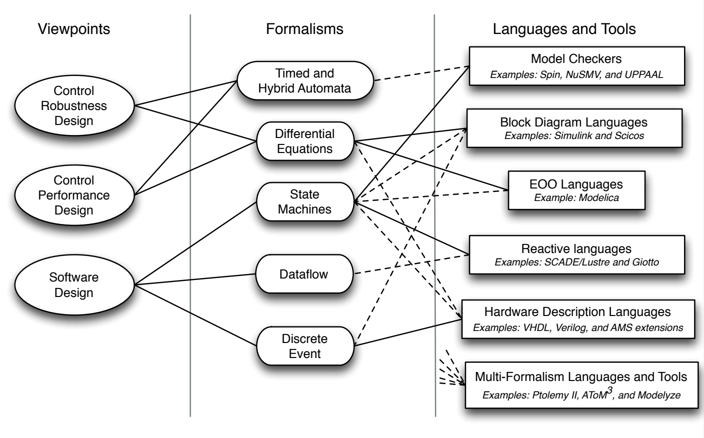
- Formalism: FSM.
- Language: the language of NuSMV.

A language can implement many formalisms.

Quiz: Which formalisms from those we have seen (DFA, FSMs, transition systems, ...) does NuSMV implement? What about Spin?

Designer's dilemmas (some of many)

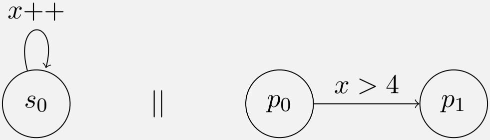
Which formalism do I need?
Which language and tool should I choose?



For a discussion, see [Broman et al., 2012].

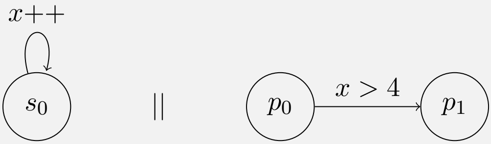
FAIRNESS

Fairness: Motivation



Will the rightmost process ever get to move to p_1?

Fairness: Motivation

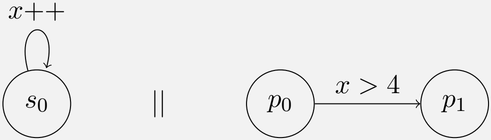


Will the rightmost process ever get to move to p_1?

Asynchronous composition transition relation:

$$\underbrace{x' = x + 1 \wedge p' = p}_{\text{from process 1}} \vee \underbrace{(x > 4 \rightarrow p' = p_1) \wedge (x \leq 4 \rightarrow p' = p_0) \wedge x' = x}_{\text{from process 2}}$$

Fairness: Motivation



Will the rightmost process ever get to move to p_1?

Asynchronous composition transition relation:

$$\underbrace{x' = x + 1 \land p' = p}_{\text{from process 1}} \lor \underbrace{(x > 4 \to p' = p_1) \land (x \leq 4 \to p' = p_0) \land x' = x}_{\text{from process 2}}$$

How to ensure that no process gets neglected forever?

Weak Fairness

If a transition is always enabled after some point on, it will eventually be taken.

Weak Fairness

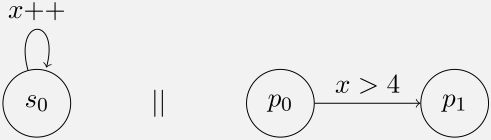
> *If a transition is always enabled after some point on, it will eventually be taken.*

or better:

> *A trace $s_0, s_1, s_2, \ldots$ is weakly unfair if there exists a transition which is enabled at all states s_i for some $i \geq K$ for some K, but never taken.*

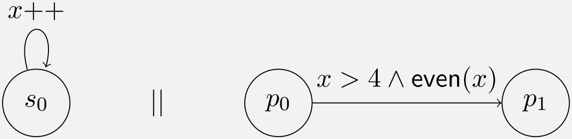
Weak Fairness

Weak fairness solves this problem:



The trace where the transition from p_0 to p_1 never happens is weakly unfair.

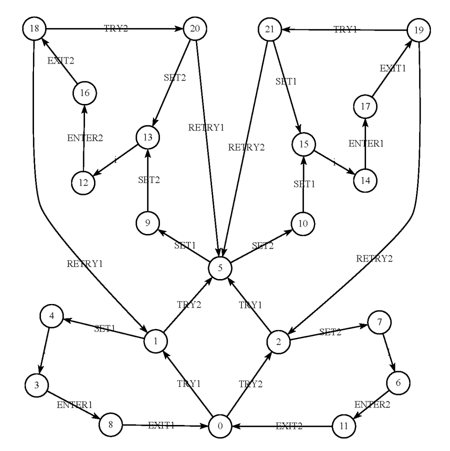
Weak Fairness is Sometimes too Weak



Here, the trace where the transition from p_0 to p_1 never happens is *not* weakly unfair, because the transition is not continually enabled.

Weak Fairness is Sometimes too Weak

More realistic application:



How to ensure that both processes eventually enter their critical section?

Strong Fairness

If a transition is infinitely-often enabled after some point on, it will eventually be taken.

or better:

A trace $s_0, s_1, s_2, ...$ *is strongly unfair if there exists a transition which is enabled infinitely often in the trace but never taken.*

Model-checking in the presence of fairness

Suppose we want to check $M \models \phi$ but it fails because some traces of M violate ϕ.

Suppose all these traces are unfair.

How to exclude them from consideration?

Model-checking in the presence of fairness

Suppose we want to check $M \models \phi$ but it fails because some traces of M violate ϕ.

Suppose all these traces are unfair.

How to exclude them from consideration?

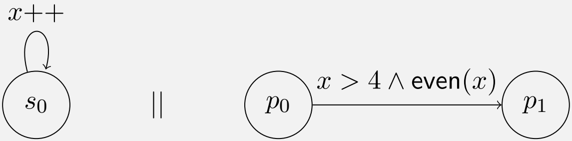
Check a different formula:

$$M \models \phi_{\mathsf{fair}} \rightarrow \phi$$

where ϕ_{fair} characterizes the fair traces.

Model-checking in the presence of fairness

Homework: Let M be the system formed by the asynchronous composition of the two processes shown below. M does not satisfy the LTL formula Fp_1. Why?



M satisfies Fp_1 if we assume strong fairness. What would ϕ_{fair} be so that

$$M \models \phi_{\mathsf{fair}} \to Fp_1?$$

Fairness not limited to asynchronous composition

Example:

```
MODULE inverter(input)
VAR
  output : boolean;
INIT
  output = FALSE
TRANS
  next(output) = !input | next(output) = output
```

This models a non-deterministic transition system.
Possible fairness requirement: if input changes, output must eventually also change.

Fairness not limited to asynchronous composition

Example:

```
MODULE inverter(input)
VAR
  output : boolean;
INIT
  output = FALSE
TRANS
  next(output) = !input | next(output) = output
```

This models a non-deterministic transition system.
Possible fairness requirement: if input changes, output must eventually also change.

Another example: a communication channel cannot keep on losing a message forever.

SUMMARY: DISCRETE SYSTEMS

Discrete Systems

- Modeling formalisms:
  - automata, state machines
  - transition systems
  - temporal logics.
- Application domain: circuits.
- Analysis and optimization algorithms:
  - state-space exploration (enumerative, symbolic)
  - bounded model-checking
  - SAT solving
  - timing analysis and retiming for circuits (today).
- Composition:
  - synchronous, asynchronous composition
  - synchronous feedback
  - contracts
  - fairness.
- Tools: Spin and NuSMV.

Next week: Dataflow

Please install:

Ptolemy II: `http://ptolemy.eecs.berkeley.edu/ptolemyII/ptII8.0/index.htm`

SDF3: `http://www.es.ele.tue.nl/sdf3/`

Bibliography

Broman, D., Lee, E., Tripakis, S., and Törngren, M. (2012).
Viewpoints, Formalisms, Languages, and Tools for Cyber-Physical Systems.
In *6th International Workshop on Multi-Paradigm Modeling (MPM'12)*.
http://www.eecs.berkeley.edu/~stavros/papers/mpm2012.pdf.

de Alfaro, L. and Henzinger, T. (2001).
Interface automata.
In *Foundations of Software Engineering (FSE)*. ACM Press.

Holzmann, G. (2003).
The Spin Model Checker.
Addison-Wesley.

Liskov, B. and Wing, J. (1994).
A behavioral notion of subtyping.
ACM Trans. Program. Lang. Syst., 16(6):1811–1841.

Manna, Z. and Pnueli, A. (1991).
The Temporal Logic of Reactive and Concurrent Systems: Specification.
Springer-Verlag, New York.

Tripakis, S., Lickly, B., Henzinger, T. A., and Lee, E. A. (2011).
A Theory of Synchronous Relational Interfaces.
ACM Transactions on Programming Languages and Systems (TOPLAS), 33(4).
Pre-print available at http://www.eecs.berkeley.edu/~stavros/papers/acm-toplas.pdf.