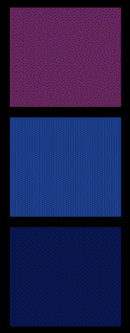
EE249 - Fall 2012
Lecture 18: Overview of Concrete Contract Theories

Alberto Sangiovanni-Vincentelli
Pierluigi Nuzzo

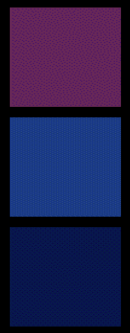
Outline: Contracts and compositional methods for system design

- Where and why using contracts?
- Introduction to contracts
- Mathematical meta-theory of contracts
- Overview of concrete contract theories



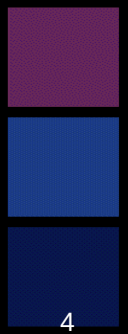
Outline: Contracts and compositional methods for system design

- Where and why using contracts?
- Introduction to contracts
- Mathematical meta-theory of contracts
- Overview of concrete contract theories
  - Assume/Guarantee contracts
  - Introduction to interface theories



Concrete contract theories

- Assume/Guarantee contracts
  - Dataflow A/G contracts
  - Synchronous A/G contracts
  - Observers
- Introduction to interface theories

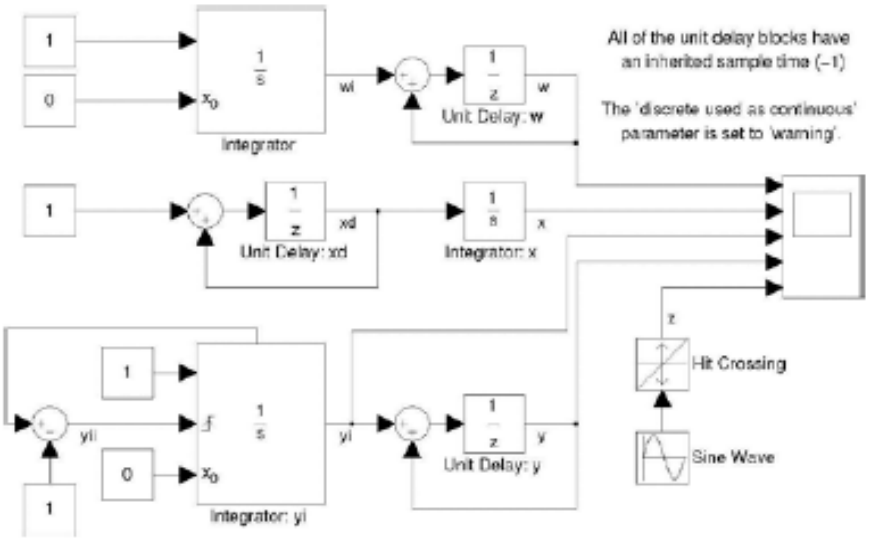


Dataflow A/G contracts

Component model

Dataflow diagram

[Simulink, Scade]



- Component *M* is an assertion *P*

$$P \subseteq \Sigma \mapsto D^* \cup D^\omega$$

  - Subset of the set of all finite or infinite behaviors over alphabet Σ
  - Assume all components have a fixed alphabet of variables, identical domain *D*
  - No global clock
- Components are always composable

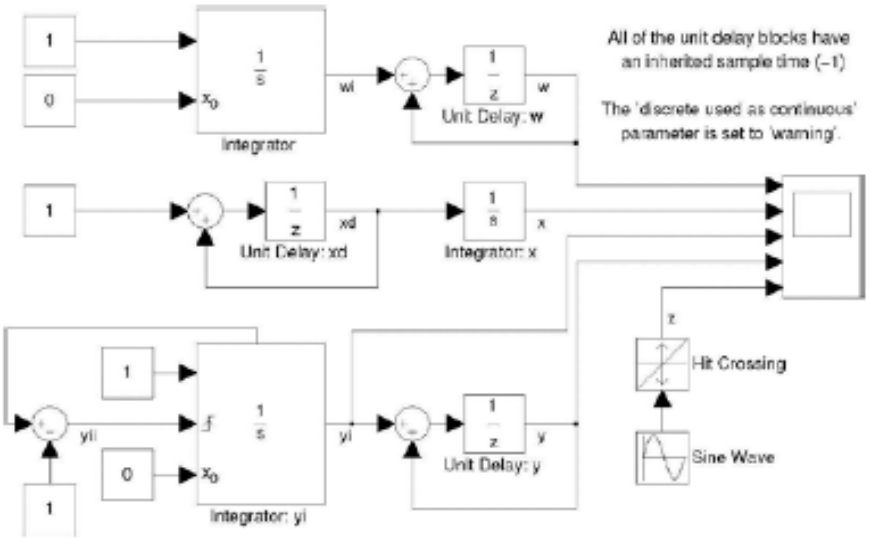
$$P_1 \times P_2 = P_1 \wedge P_2$$

Dataflow A/G contracts

Component model

Dataflow diagram

[Simulink, Scade]



Source: A. Benveniste

Contract model

Assume/Guarantee contracts

[Benveniste & al]

C = (A,G)

= (assumption,guarantee)

- A = assertion (diagram)
- G = assertion (diagram)

(E,M) ⊨ C

1. E satisfies A
2. E×M satisfies G

What is guaranteed is "A⇒G"

Dataflow A/G contracts: saturated form

- $M_C = G \vee \neg A$ is the maximal implementation
- Contract in saturated form: *(A, G')* such that $G \supseteq \neg A$ or $G \vee A = T$
  - Consistent iff $G \neq \phi$
  - Compatible iff $A \neq \phi$
- Any contract C = *(A,G)* is equivalent (identical sets of implementations) to a contract in saturated form
  - Take $G' = G \vee \neg A$

Dataflow A/G contracts: operations

- Consider C, C', C_1, C_2 saturated contracts with identical input/output alphabets
  - Refinement

$$\mathcal{C}' \preceq \mathcal{C} \text{ holds iff } \begin{cases} A' \geq A \\ G' \leq G \end{cases}$$

  - Conjunction

$$\mathcal{C}_1 \wedge \mathcal{C}_2 = (A_1 \vee A_2, G_1 \wedge G_2)$$

  - Composition

$$\begin{aligned} G &= G_1 \wedge G_2 \\ A &= \max \left\{ A \;\middle|\; \begin{array}{l} A \wedge G_2 \Rightarrow A_1 \\ \text{and} \\ A \wedge G_1 \Rightarrow A_2 \end{array} \right\} \end{aligned} \rightarrow \begin{aligned} G &= G_1 \wedge G_2 \\ A &= (A_1 \wedge A_2) \vee \neg(G_1 \wedge G_2) \end{aligned}$$

Dataflow A/G contract extensions: inputs and outputs

- Decompose alphabet into inputs and outputs

$$\Sigma = \Sigma^{\text{in}} \cup \Sigma^{\text{out}}$$

- A component is a tuple $M = (\Sigma^{\text{in}}, \Sigma^{\text{out}}, P)$
- M_1 and M_2 composable if $\Sigma_1^{\text{out}} \cap \Sigma_2^{\text{out}} = \emptyset$
  - Then $\Sigma^{\text{out}} = \Sigma_1^{\text{out}} \cup \Sigma_2^{\text{out}}$ $\quad \Sigma^{\text{in}} = \Sigma - \Sigma^{\text{out}}$ $\quad P = P_1 \cap P_2$
- Useful to include exceptions: undesirable events caused by the component itself (and not by its inputs)
  - Capture with a special element: *fail* $\in D$, e.g. *x/0* := *fail* for any *x*
  - M is “free of exceptions” if it accepts all inputs and does not cause by itself the occurrence of fail in its behaviors

Dataflow A/G contract extensions: inputs and outputs

- A contract is a tuple $\mathcal{C} = (\Sigma^{\text{in}}, \Sigma^{\text{out}}, A, G)$
- Legal environment for C are all "free of exception" components E such that $\Sigma_E^{\text{in}} = \Sigma_{\mathcal{C}}^{\text{out}}$, $\Sigma_E^{\text{out}} = \Sigma_{\mathcal{C}}^{\text{in}}$ and $P_E \subseteq A$
- Set of all implementations for C are all components *M* free of exceptions and such that
  - $\Sigma_M^{\text{in}} = \Sigma_{\mathcal{C}}^{\text{in}}$
  - $\Sigma_M^{\text{out}} = \Sigma_{\mathcal{C}}^{\text{out}}$
  - $P_{E \times M} \subseteq G$ for every legal environment *E* of C

Dataflow A/G contract extensions: variable alphabets

- The actual alphabet of the complete design is not known in advance
  - Contracts and components have their own alphabet
  - All relations and operations can be extended after alphabet equalization
- Given *P* assertion over Σ and Σ' ⊆ Σ, define
  - *Projection* **pr** $_{\Sigma'}$(*P*) over Σ': the set of all restrictions to Σ' of all behaviors belonging to *P*
  - *Inverse projection* **pr**$^{-1}$ $_{\Sigma''}$(*P*) over Σ'' ⊇ Σ: the set of all behaviors over Σ'' projecting, to Σ, as behaviors of *P*
  - *Alphabet equalization* of (Σ_1, P_1) and (Σ_2, P_2) the two assertions (Σ, **pr**$^{-1}$ $_{\Sigma}$(P_i)), *i*=1,2 where $\Sigma = \Sigma_1 \cup \Sigma_2$

Synchronous A/G contracts

- Synchronous A/G contracts can be obtained as a variant of dataflow contracts by redefining assertions (set of behaviors) as

$$P \quad \subseteq \quad (\Sigma \mapsto D)^* \cup (\Sigma \mapsto D)^\omega$$

  - Behaviors are finite or infinite sequences of reactions, which are the assignment of a value to each symbol of the alphabet (synchronous model)
  - Compare with dataflow, where behaviors are tuples of finite or infinite flows, one for each symbol of the alphabet

Observers for A/G contracts

- Assertion P defines a verdict by setting

$$b_P(\sigma) = \mathrm{T} \quad \text{if and only if } \sigma \in P$$

- To return verdict in some finite amount of time, on-line interpretations can be defined as follows:
  - $b_C^E(E)$ is performed by drawing non-deterministically a behavior σ of E and then evaluating $b_A(\sigma)$
  - $b_C^M(M)$ is performed by drawing non-deterministically a behavior σ of M and then evaluating $b_A(\sigma) \Longrightarrow b_G(\sigma)$
- Lemma 3 for generic observers specializes to an effective semi-decision procedure
  - If b_A outputs F then C is incompatible
  - If $b_A \Longrightarrow b_G$ outputs F then C is inconsistent

A/G contracts: discussion

- A/G contracts are "flexible": can accommodate different models of communication and exceptions
  - E.g. extended to real-time interfaces, analog mixed-signal integrated circuit design, stochastic models
- Very natural for formalizing requirements
  - Requirements can be naturally seen as assertions and categorized as assumptions and guarantees
- Suffers from the need to manipulate negations of assertions
  - Generally not effective, except for finite state automata and finite alphabets of actions
  - Using observers or abstract interpretation can partially mitigate this

A/G contract variant: circular reasoning

- Replace parallel composition to achieve separate development of components
  - Overcome the problems of certain models with computation of the operators
  - When it is sound, circular reasoning allows checking relations between composite contracts based on their components
- Simplest form
  - If design M satisfies property G under assumption A
  - If design N satisfies assumption A
  - Then M×N satisfies G
- Seemingly circular arguments are justified for assumptions and guarantees expressed as safety properties
  - can be proven or disproved by finite observations
  - need to observe restrictions on how A/G refer to ports
  - Restricted notions of refinement under context can be required for circular reasoning to hold

Concrete contract theories

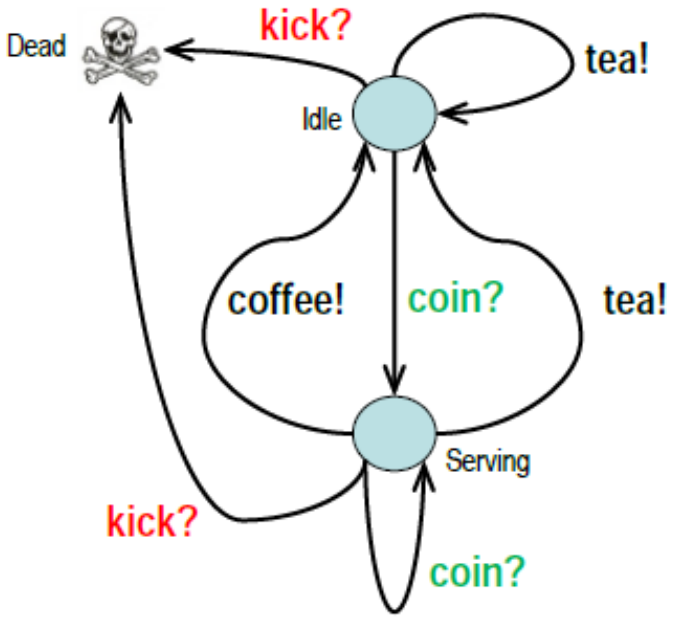
- Assume/Guarantee contracts
- Introduction to interface theories
  - Components as i/o automata
  - Interface automata
  - Modal interfaces
  - Application example: a parking garage

Interface theories

Component model

Input-output Automata

[Lynch]



- A tuple $M = (\Sigma^{\text{in}}, \Sigma^{\text{out}}, Q, q_0, \rightarrow)$
  - Σ^{in} and Σ^{out} are disjoint finite input and output alphabets (Σ is their union)
  - Q is a finite set of states and $q_0 \in Q$ is the initial state
  - $\rightarrow \subseteq Q \times \Sigma \times Q$ is the transition relation
  - $\rightarrow$ extends by concatenation to $\rightarrow^* \subseteq Q \times \Sigma^* \times Q$

Source: A. Benveniste

I/O-Automata: definitions

- A state *q'* is reachable from *q*
  - there exists some word *m* such that $q \xrightarrow{m} * q'$
- Restrict to *deterministic automata*, such that
  - $q \xrightarrow{\alpha} q_1$ and $q \xrightarrow{\alpha} q_2$ implies $q_2 = q_1$
- M_1 and M_2 with identical alphabet are composable if $\Sigma_1^{\text{out}} \cap \Sigma_2^{\text{out}} = \emptyset$

$$M = M_1 \times M_2 \quad \begin{cases} \Sigma^{\text{out}} = \Sigma_1^{\text{out}} \cup \Sigma_2^{\text{out}} & \Sigma^{\text{in}} = \Sigma_1^{\text{in}} \cap \Sigma_2^{\text{in}} \\ Q = Q_1 \times Q_2 \text{ and } q_0 = (q_{1,0}, q_{2,0}) \\ (q_1, q_2) \xrightarrow{\alpha} (q_1', q_2') \text{ iff } q_i \xrightarrow{\alpha}_i q_i', i = 1, 2 \end{cases}$$

I/O-Automata: definitions

- *M* is *closed* when $\Sigma^{in} = \emptyset$ and *open* otherwise
- For M_1 and M_2 and two of their states q_1 and q_2, say that q_1 simulates q_2 ($q_2 \leq q_1$) if

$$\forall \alpha, q_2' \text{ such that } q_2 \xrightarrow{\alpha}_2 q_2' \Rightarrow \begin{cases} q_1 \xrightarrow{\alpha}_1 q_1' \\ \text{and } q_2' \leq q_1' \end{cases}$$

- M_1 simulates M_2 ($M_2 \leq M_1$), if $q_{2,0} \leq q_{1,0}$
- Variable alphabet is handled with *alphabet equalization*
  - For $\Sigma' \supset \Sigma$ define $M^{\uparrow \Sigma'} = (\Sigma^{in} \cup (\Sigma' \setminus \Sigma), \Sigma^{out}, Q, q_0, \rightarrow')$
  - The new transition is obtained by adding, for each state and each added action, a self-loop at this state labeled with this action

I/O-Automata: definitions

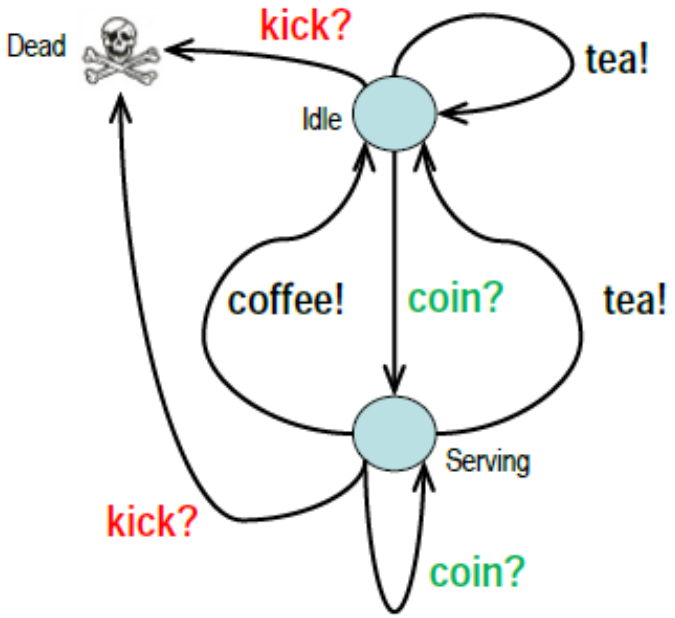
$$M \text{ is } \textit{receptive} \text{ iff } \forall q \in Q, \forall \alpha \in \Sigma^{\text{in}} : \; q \xrightarrow{\alpha}$$

- Components and environments are *receptive* (or input enabled) i/o-automata for use in interface theories, i.e. they should react by proper response to any input stimulus in any state
- Receptiveness is stable under parallel composition
- Interface automata can be interpreted as contracts for i/o-automata (based on the meta-theory)
  - Present interface automata for the case of *fixed alphabet* to simplify
  - Begin with the definition of interface automata which are possibly non-receptive i/o-automata and interpret as contracts

Interface theories

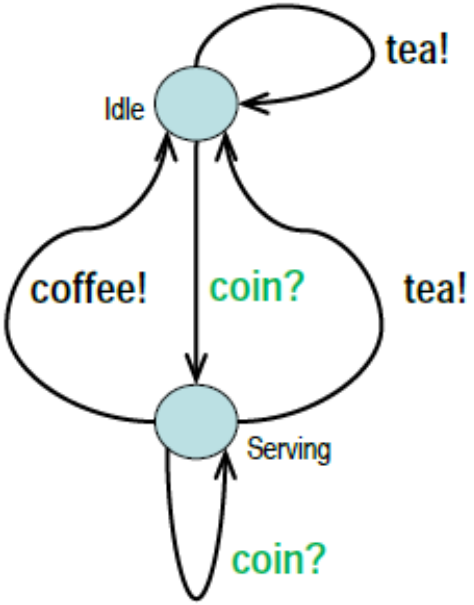
Component model

Input-output Automata
[Lynch]



Contract model

Interface Automata
[de Alfaro Henzinger]



Source: A. Benveniste

Interface automata as contracts

- An Interface Automaton is a tuple $C = (\Sigma^{\text{in}}, \Sigma^{\text{out}}, Q, q_0, \rightarrow)$
  - Σ^{in}, Σ^{out}, Q, q_0 and → are as in i/o-automata
  - The initial state q_0 may not be in Q
  - If $q_0 \in Q$ holds, C defines a contract
- For a contract C in the form of interface automaton, the set $\mathcal{E}_C$ of the legal environments for C collects all components E such that:
  - $\Sigma_E^{\text{in}} = \Sigma^{\text{out}}$, $\Sigma_E^{\text{out}} = \Sigma^{\text{in}}$, i.e. E and C (as i/o-automata) are composable and $E \times C$ is closed
  - For any output action $\alpha \in \Sigma^{\text{in}}$ of environment E such that $q_E \xrightarrow{\alpha} E$ and any reachable state (q_E, q) of $E \times C$, then $(q_E, q) \xrightarrow{\alpha}_{E \times C}$ holds.

Interface automata as contracts: definition

- There exists a unique *maximal environment* E_C, i.e. $E_C \times C$ simulates $E \times C$ for any E in the environment set:

$$E_C = (\Sigma^{\text{out}}, \Sigma^{\text{in}}, Q \cup \{\top\}, q_0, \rightarrow_{E_C})$$

where:

(a) the restriction to Q×Σ×Q of the transition relation $\rightarrow_{E_C}$ coincides with →

(b) For each $q \in Q$: $q \xrightarrow{\alpha}_{E_C} \top$ holds if and only if $\neg[q \xrightarrow{\alpha}]$ and $\alpha \in \Sigma^{\text{out}}$

(c) $\top \xrightarrow{\alpha}_{E_C} \top$ holds for any $\alpha \in \Sigma$

Interface automata as contracts: definition

- The set $\mathcal{M}_C$ of the implementations of C collects all components M such that i/o-automaton C simulates $E_C \times M$
- Lemma: $q_0 \in Q$ is the necessary and sufficient condition for C to be both consistent and compatible in the sense of the meta-theory, i.e. $\mathcal{E}_C$ and $\mathcal{M}_C$ are non-empty

Interface automata: refinement and conjunction

- Contract refinement is equivalent to alternate simulation
- No simple formula for the conjunction of contracts is known (just a few recent results in this direction…)

Interface automata: alternate simulation

- For C_i, i=1,2, two contracts, say that two respective states q_i, i=1,2, are in alternate simulation, written $q_2 \preceq q_1$, if

$$\forall \alpha \in \Sigma_2^{\text{out}}, q_2' \text{ s.t. } q_2 \xrightarrow{\alpha}_2 q_2' \quad \Rightarrow \quad \begin{cases} \alpha \in \Sigma_1^{\text{out}} \\ q_1 \xrightarrow{\alpha}_1 q_1' \\ q_2' \preceq q_1' \end{cases}$$

$$\forall \alpha \in \Sigma_1^{\text{in}}, q_1' \text{ s.t. } q_1 \xrightarrow{\alpha}_1 q_1' \quad \Rightarrow \quad \begin{cases} \alpha \in \Sigma_2^{\text{in}} \\ q_2 \xrightarrow{\alpha}_2 q_2' \\ q_2' \preceq q_1' \end{cases}$$

- Say that C_2 refines C_1, written $C_2 \preceq C_1$ if $q_{2,0} \preceq q_{1,0}$
- Alternate simulation can be effectively checked

Interface automata: parallel composition

- Given C_1 and C_2 i/o-automata, $C_2 \times C_1$ is a first candidate
  - Does not work because of the condition involving environments in the meta-theory

$$E \models^{\mathrm{E}} \mathcal{C} \Rightarrow [E \times M_2 \models^{\mathrm{E}} C_1 \text{ and } E \times M_1 \models^{\mathrm{E}} C_2]$$

  - This requires

$$\forall \alpha \in \Sigma_i^{\mathrm{out}}\colon q_i \xrightarrow{\alpha}_i \;\Rightarrow\; (q_1, q_2) \xrightarrow{\alpha}_{C_2 \times C_1}$$

    otherwise pairs *(q_1, q_2)* are called illegal and must be pruned away

  - *(q_1, q_2)* is illegal when one IA wants to submit an output whereas the other one refuses to accept it (i.e. a mismatch of assumptions and guarantees)

Interface automata: discussion

- Pruning away illegal pairs together with corresponding incoming transitions may cause other illegal pairs to occur
- These must also be pruned away until a fixpoint is reached
- If the resulting contract has empty set of states, then the composition of the interfaces is inconsistent and incompatible
- Variable alphabets can be handled by using alphabet equalization via inverse projections
  - Correct for composition, …but not clear for conjunction
- Conjunction and alphabet equalization elegantly addressed by Modal Interfaces, which adopt different extension procedures for composition and conjunction
  - Given a system level contract C as a modal interface and a topological architecture, there are algorithms to automatically refine C into a compatible composition of local sub-contracts (modal interfaces)

Modalities and modal specifications

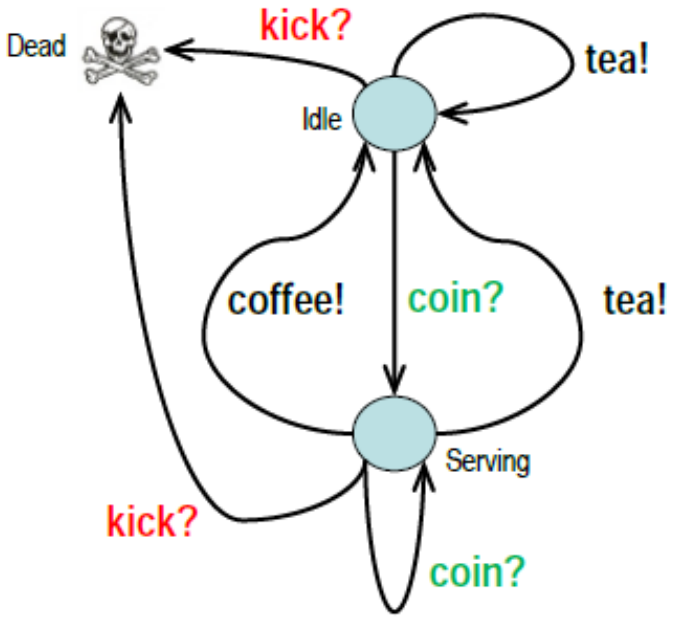
- Modal interfaces combine interface automata and modal specifications
- Modal specifications provide a framework to express both allowed and mandatory properties
  - Properties expressed as sets of traces can only specify what is forbidden
  - In interface automata it is not possible to express mandatory behaviors for designs unless time is explicitly invoked in the properties
- Assign a modality *may* or *must* to each possible transition of a system
  - A *must* transition is available in every component that implements the modal specification
  - A *may* transition needs not be
  - Every required transition is also allowed, i.e. **must** $\subseteq$ **may** (sets of transitions)

Interface theories

Component model

Input-output Automata

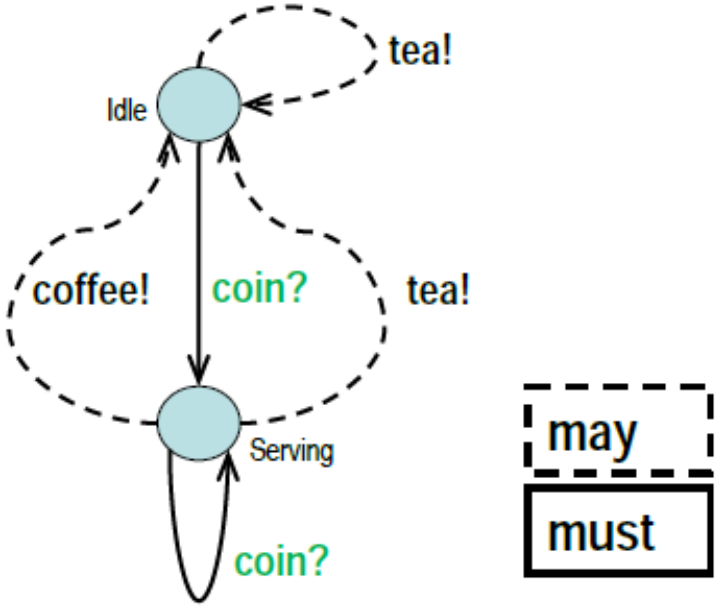
[Lynch]



Contract model

Modal Interfaces

[Larsen & al] [Raclet & al]



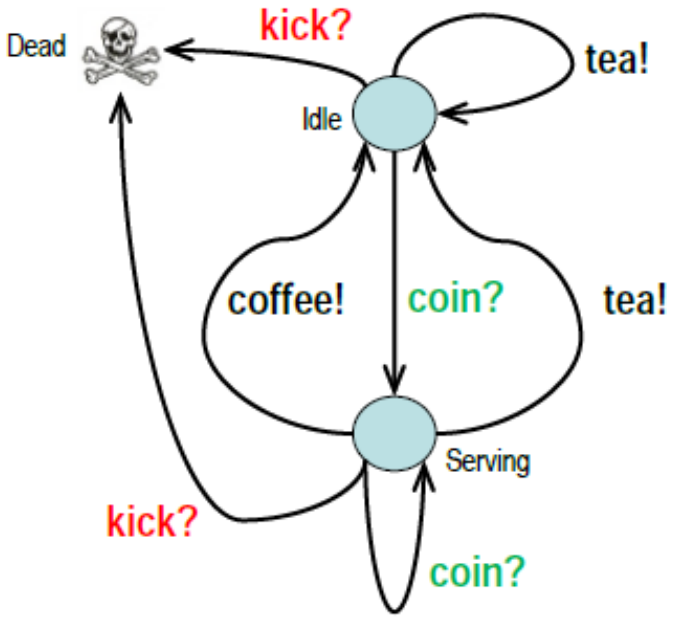
Source: A. Benveniste

Interface theories

Component model

Input-output Automata

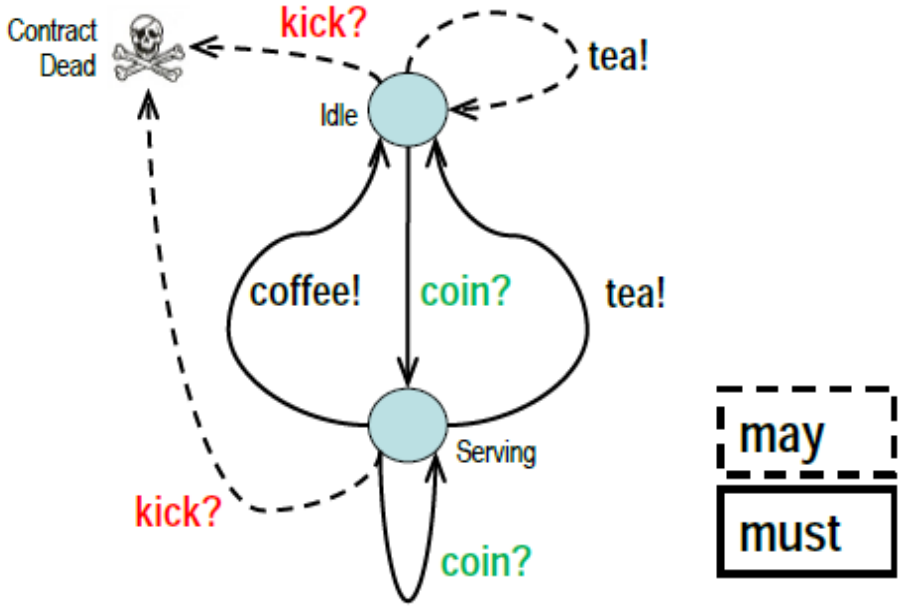
[Lynch]



Contract model

Modal Interfaces

[Larsen & al] [Raclet & al]



Source: A. Benveniste

Modal interfaces as contracts

- A Modal Interface is a tuple $\mathcal{C} = (\Sigma^{\text{in}}, \Sigma^{\text{out}}, Q, q_0, \rightarrow, \dashrightarrow)$
  - Σ^{in}, Σ^{out}, Q, q_0 are as in i/o-automata and $\rightarrow$, $\dashrightarrow$ are the *must* and *may* transition relations
  - If $q_0 \in Q$, $\mathcal{C}$ induces 2 (possibly non-receptive) i/o-automata

$$\begin{aligned} \mathcal{C}^{must} &= (\Sigma^{\text{in}}, \Sigma^{\text{out}}, Q, q_0, \rightarrow) \\ \mathcal{C}^{may} &= (\Sigma^{\text{in}}, \Sigma^{\text{out}}, Q, q_0, \dashrightarrow) \end{aligned}$$

- A modal interface is deterministic if $\mathcal{C}^{may}$ is deterministic
- $\mathcal{C}$ defines a contract where the set $\mathcal{E}_{\mathcal{C}}$ of the legal environments for $\mathcal{C}$ collects all components E such that:
  - $\Sigma_E^{\text{in}} = \Sigma^{\text{out}}$, $\Sigma_E^{\text{out}} = \Sigma^{\text{in}}$, i.e. E and $\mathcal{C}^{must}$ ($\mathcal{C}^{may}$) are composable and $E \times \mathcal{C}^{must}$ ($E \times \mathcal{C}^{may}$) is closed
  - For any output action $\alpha \in \Sigma^{\text{in}}$ such that $q_E \xrightarrow{\alpha}_E$ and any reachable state (q_E, q) of $E \times \mathcal{C}^{may}$, then $(q_E, q) \xrightarrow{\alpha}_{E \times \mathcal{C}^{must}}$

Modal Interface: definition

- There exists a unique *maximal environment* E_C, i.e. $E_C \times C^{may}$ simulates $E \times C^{may}$ for any E in the environment set:

$$E_{\mathcal{C}} = (\Sigma^{\text{out}}, \Sigma^{\text{in}}, Q \cup \{\top\}, q_0, \rightarrow_{E_{\mathcal{C}}})$$

where:

(a) the restriction to $Q \times \Sigma^{\text{in}} \times Q$ of the transition relation $\rightarrow_{E_{\mathcal{C}}}$ coincides with $\rightarrow$; the restriction to $Q \times \Sigma^{\text{out}} \times Q$ of the transition relation $\rightarrow_{E_{\mathcal{C}}}$ coincides with $\dashrightarrow$

(b) For any $q \in Q$, $q \xrightarrow{\alpha}_{E_{\mathcal{C}}} \top$ holds if and only if $\neg[q \overset{\alpha}{\dashrightarrow}]$ and $\alpha \in \Sigma^{\text{out}}$

(c) $\top \xrightarrow{\alpha}_{E_{\mathcal{C}}} \top$ holds for any $\alpha \in \Sigma$

Modal interface automata as contracts: definition

- The set $\mathcal{M}_C$ of the implementations of C collects all components M such that:
- $E_C \times C^{may}$ simulates $E_C \times M$ [only **may** transitions are allowed for $E_C \times M$]
- For any reachable state (q_E, q_M) of $E_C \times M$ and (q_E, q) of $E_C \times C^{may}$ such that $(q_E, q_M) \leq (q_E, q)$ and any action $\alpha \in \Sigma_M^{out}$ such that $q \xrightarrow{\alpha}$, then $(q_E, q_M) \xrightarrow{\alpha}_{E_C \times M}$ [**must** transitions are mandatory in $E_C \times M$]
- Consistency and compatibility. State q is *consistent* if $q \xrightarrow{\alpha}$ implies $q \dashrightarrow$, otherwise q is *inconsistent*
  - Inconsistent states can be pruned away without changing $\mathcal{M}_C$
  - At the fixpoint, Q_{con} only collects consistent states: C is consistent and compatible if and only if $q_0 \in Q_{con}$

Modal refinement and contracts

- Contract refinement is instantiated by modal refinement
- Given q state of a modal interface (MI), introduce the sets:

$$\begin{aligned} may(q) &= \{\alpha \in \Sigma \mid q \overset{\alpha}{\dashrightarrow} \} \\ must(q) &= \{\alpha \in \Sigma \mid q \overset{\alpha}{\rightarrow} \} \end{aligned}$$

- For C_i, i=1,2, two MI, and q_i, i=1,2, two respective states, say that q_2 refines q_1, written $q_2 \preceq q_1$, if

$$\begin{cases} may_2(q_2) \subseteq may_1(q_1) \\ must_2(q_2) \supseteq must_1(q_1) \end{cases}$$

$$and\ \forall \alpha \in \Sigma : \begin{cases} q_1 \overset{\alpha}{\dashrightarrow}_1 q_1' \\ q_2 \overset{\alpha}{\dashrightarrow}_2 q_2' \end{cases} \implies q_2' \preceq q_1'$$

- Say that C_2 refines C_1, written $C_2 \preceq C_1$, if $q_{2,0} \preceq q_{1,0}$
- Contract refinement for (C_2, C_1) is equivalent to modal refinement for $([C_2],[C_1])$ obtained after pruning away all inconsistent states

Modal interfaces: conjunction and composition

- Conjunction: GLB for the refinement order
  - Compute a pre-conjunction of two MI: $\Sigma^{in} = \Sigma_1^{in} = \Sigma_2^{in}$, $\Sigma^{out} = \Sigma_1^{out} = \Sigma_2^{out}$, $Q = Q_1 \times Q_2$, $q_0 = (q_{1,0}, q_{2,0})$ and

$$\begin{aligned} must(q_1, q_2) &= must_1(q_1) \cup must_2(q_2) \\ may(q_1, q_2) &= may_1(q_1) \cap may_2(q_2) \end{aligned}$$

  - Prune away inconsistent states
- Composition is also defined in two steps:
  - Compute a pre-composition of two (composable) MI: $\Sigma^{in} = \Sigma_1^{in} = \Sigma_2^{in}$, $\Sigma^{out} = \Sigma_1^{out} \cup \Sigma_2^{out}$, $Q = Q_1 \times Q_2$, $q_0 = (q_{1,0}, q_{2,0})$ and

$$\begin{aligned} must(q_1, q_2) &= must_1(q_1) \cap must_2(q_2) \\ may(q_1, q_2) &= may_1(q_1) \cap may_2(q_2) \end{aligned}$$

  - Prune away illegal states (an effective procedure is available), i.e. (q_1, q_2) such that

$$\begin{aligned} may_1(q_1) \cap \Sigma_2^{in} &\not\subseteq must_2(q_2) \\ may_2(q_2) \cap \Sigma_1^{in} &\not\subseteq must_1(q_1) \end{aligned}$$

Other theories…

Component model	Contract model
Systems with non-trivial data	Observers, Abstract Interpretation
Timed systems	Restrictions needed
Probabilistic systems (reliability)	In progress

The Parking Garage toy example

- Top-level specification: requirements document
  - generic gate
  - payment machine
  - supervisor
- Different formalisms are used (textual and automata)
- Responsibilities assigned for each requirement { assumption / guarantee
- Requirements are translated into Modal Interfaces

Source: A. Benveniste

The Parking Garage toy example

- Top-level specification: requirements document
  - generic gate
  - payment machine
  - supervisor
- Different formalisms are used (textual and automata)
- Responsibilities assigned for each requirement { assumption / guarantee
- Requirements are translated into Modal Interfaces

- Consistency, Compatibility, Correctness, Completeness
- Top-level Requirements ($\wedge$) $\longrightarrow$ Architecture of sub-systems ($\otimes$)
- Each sub-system can be submitted to a different supplier

Source: A. Benveniste

Top level requirements

Top-level specification: assumptions & guarantees

gate(x) where $x \in$ {entry, exit}

- $R_{g.1}(x)$: "vehicles shall not pass when x_gate is closed",
- $R_{g.2}(x)$: after ?vehicle_pass ?vehicle_pass is forbidden
- $R_{g.3}$: after !x_gate_open !x_gate_open is forbidden and after !x_gate_close !x_gate_close is forbidden

payment

- $R_{p.1}$: "user inserts a coin every time a ticket is inserted and only then"
- $R_{p.2}$: "user may insert a ticket only initially or after an exit ticket has been issued"
- $R_{p.3}$: "exit ticket is issued after ticket is inserted and payment is made and only then"

supervisor

- $R_{g.1}$(entry)
- $R_{g.1}$(exit)
- $R_{g.2}$(entry)
- $R_{g.2}$(exit)
- $R_{s.1}$: initially and after !entry_gate close !entry_gate open is forbidden
- $R_{s.2}$: after !ticket_issued !entry_gate open must be enabled
- $R_{s.3}$: "at most one ticket is issued per vehicle entering the parking and tickets can be issued only if requested and ticket is issued only if the parking is not full"
- $R_{s.4}$: "when the entry gate is closed, the entry gate may not open unless a ticket has been issued"
- $R_{s.5}$: "the entry gate must open when a ticket is issued"
- $R_{s.6}$: "exit gate must open after an exit ticket is inserted and only then"
- $R_{s.7}$: "exit gate closes only after vehicle has exited parking"

Source: A. Benveniste

Formalizing requirements as contracts

Top-level specification

gate(x) where $x \in$ {entry, exit}

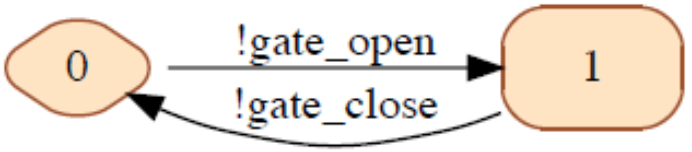
$R_{g.1}(x)$: "vehicles shall not pass when x_gate is closed",

$R_{g.2}(x)$: after ?vehicle_pass ?vehicle_pass is forbidden

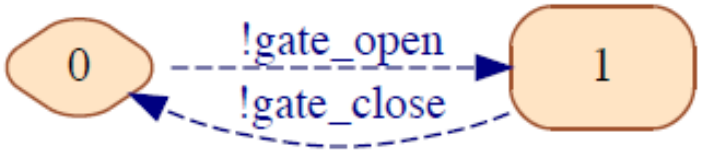
$R_{g.3}$: after !x_gate_open !x_gate_open is forbidden and after !x_gate_close !x_gate_close is forbidden

Translating the guarantees:

$R_{g.3}$ as an i/o-automaton:



$R_{g.3}$ as a Modal Interface:



Source: A. Benveniste

Formalizing requirements as contracts

Top-level specification

gate(x) where $x \in$ {entry, exit}

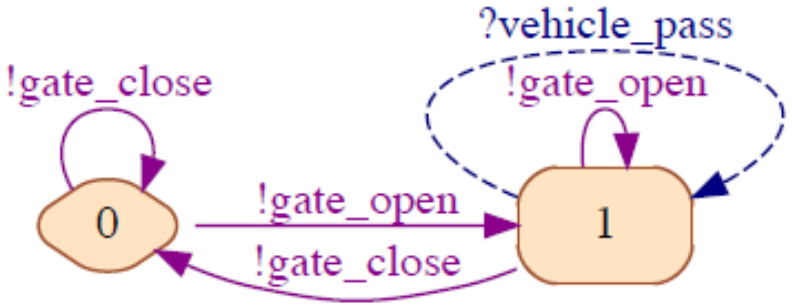
$R_{g.1}(x)$: "vehicles shall not pass when x_gate is closed",

$R_{g.2}(x)$: after ?vehicle_pass ?vehicle_pass is forbidden

$R_{g.3}$: after !x_gate_open !x_gate_open is forbidden and after !x_gate_close !x_gate_close is forbidden

Translating the assumptions:

$R_{g.1}(x)$:



Source: A. Benveniste

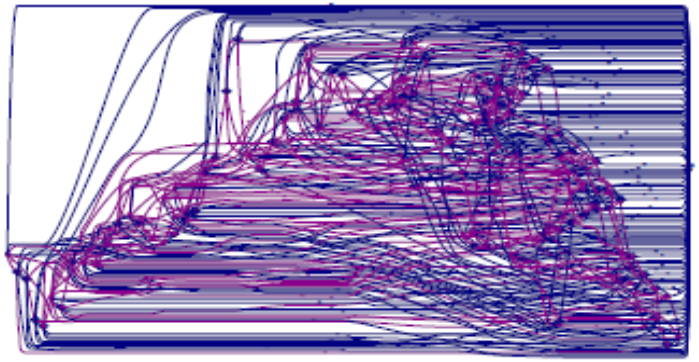
Functional specification: combining chapters

Top-level specification: $\mathcal{C} = \mathcal{C}_{\text{gate}} \wedge \mathcal{C}_{\text{payment}} \wedge \mathcal{C}_{\text{supervisor}}$

gate
payment
supervisor

- $R_{g.1}$(entry)
- $R_{g.1}$(exit)
- $R_{g.2}$(entry)
- $R_{g.2}$(exit)
- $R_{s.1}$:initially and after !entry_gate close !entry_gate open is forbidden
- $R_{s.2}$:after !ticket_issued !entry_gate open must be enabled
- $R_{s.3}$:"at most one ticket is issued per vehicle entering the parking and tickets can be issued only if requested and ticket is issued only if the parking is not full"
- $R_{s.4}$:"when the entry gate is closed, the entry gate may not open unless a ticket has been issued"
- $R_{s.5}$:"the entry gate must open when a ticket is issued"
- $R_{s.6}$:"exit gate must open after an exit ticket is inserted and only then"
- $R_{s.7}$:"exit gate closes only after vehicle has exited parking"

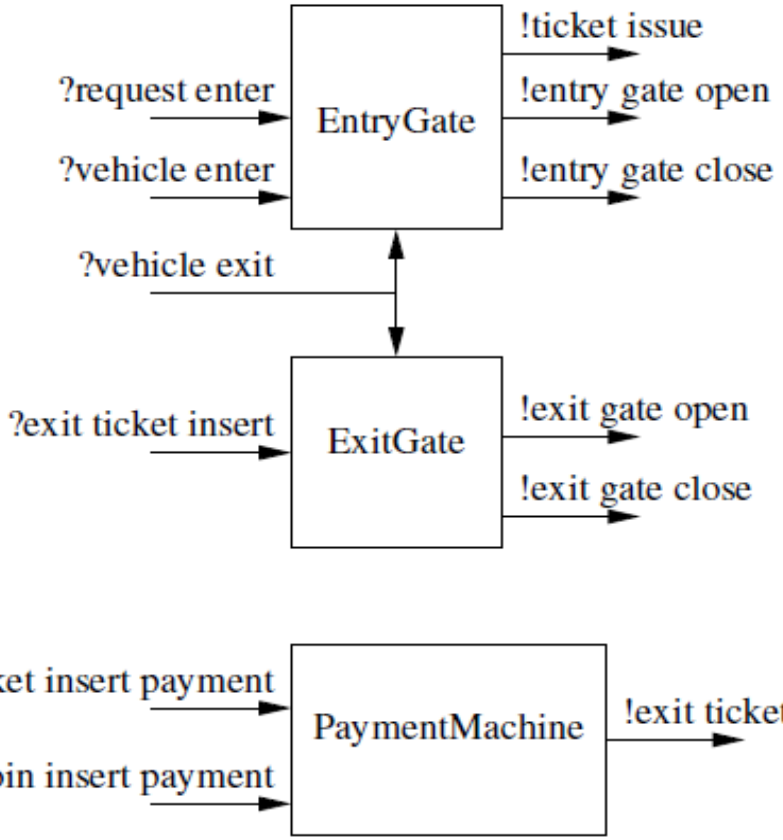
$\mathcal{C}_{\text{supervisor}}$:



Source: A. Benveniste

Subcontracting to suppliers

Architecture for sub-contracting $\mathcal{C}$ as a $\otimes$-composition of sub-systems (this is the duty of the designer)

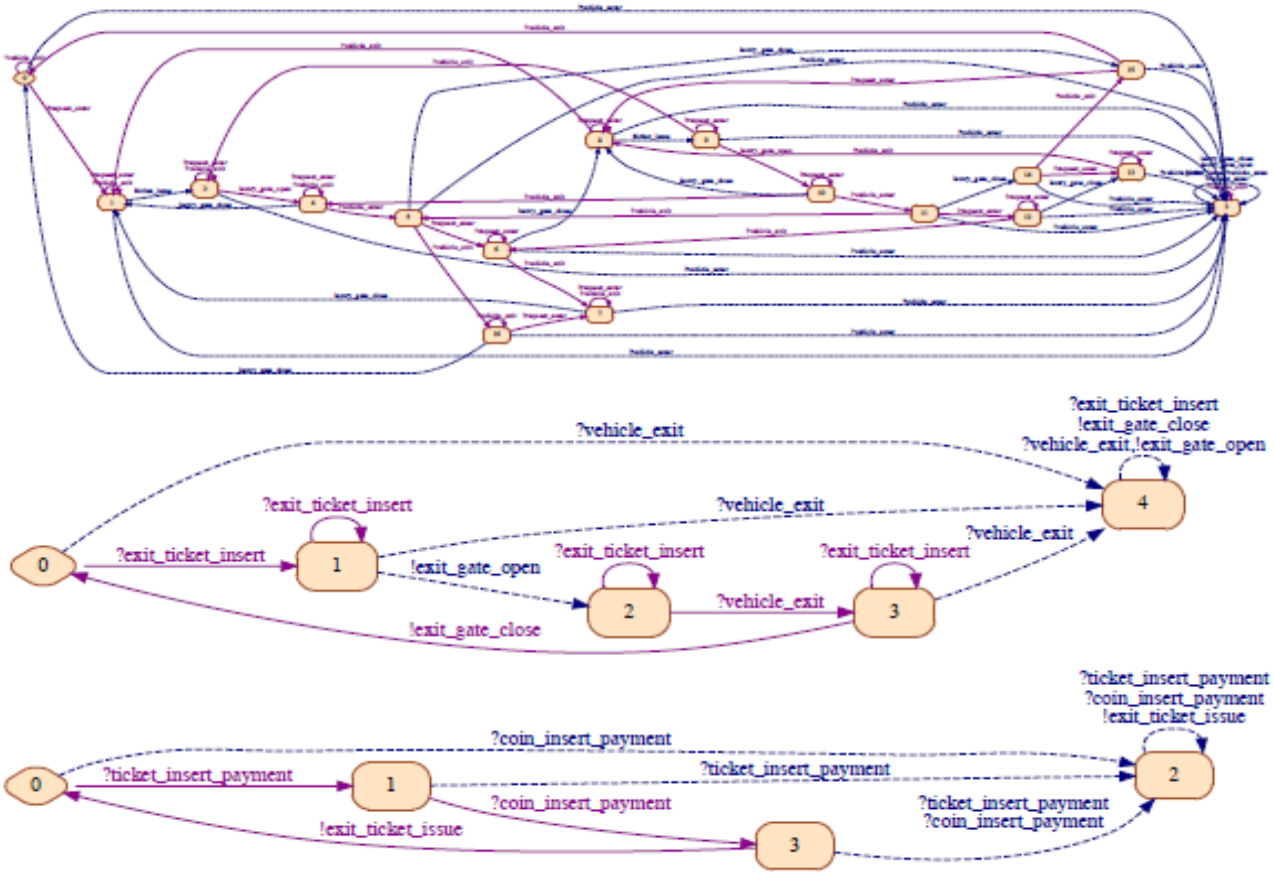


observe that the supervisor is distributed among the gates

Source: A. Benveniste

Automatic decomposition

The following $\otimes$-decomposition of $\mathcal{C}$ into three sub-contracts was automatically generated (note the reduction in size)



Source: A. Benveniste