Multiple viewpoints contracts and residuation

presented by Albert Benveniste

Jean-Baptiste Raclet – INRIA Rennes, now with INRIA Grenoble
Eric Badouel, Albert Benveniste, Benoît Caillaud – INRIA Rennes
Roberto Passerone – PARADES / University of Trento
Tom Henzinger – EPFL

2nd International Workshop on Foundations of Interface Technologies
ETAPS, Budapest, April 5, 2008

Motivations

Embedded systems possess many components + different viewpoints

- Function
- Timing
- Reliability
- Energy
- QoS
- …

- The designer may want to:
  - consider all viewpoints for each component
  - implement each component
  - compose the implementations

- Alternatively she may want to consider viewpoints incrementally:
  - consider all viewpoints for each component except Safety + QoS
  - implement each component
  - compose the implementations
  - Revisit her design for safety and QoS, possibly with a different, coarser grain, architecture

Embedded systems possess many components + different viewpoints

Combining contracts for the different viewpoints of a same component

$\neq$

Combining contracts for different components

- The designer may want to:
  - consider all viewpoints for each component
  - implement each component
  - compose the implementations
- Alternatively she may want to consider viewpoints incrementally:
  - consider all viewpoints for each component except Safety + QoS
  - implement each component
  - compose the implementations
  - Revisit her design for safety and QoS, possibly with a different, coarser grain, architecture
- **Is it a problem? Yes, it is…**

What is the problem?

- Works on Assume/Guarantee reasoning [Hoare, Gries, Lamport, Meyer, Back & von Wright] and Interfaces [deAlfaro-Henzinger 01] have well addressed the following central issues:
  - The principle of substituability and its consequences on the covariant/contravariant nature of refinement – assumptions must get weaker and guarantees stronger
  - How to compose contracts for different components
  - How to actually represent a contract in a computationally effective way, *e.g.,* the ***interface automata*** by deAlfaro-Henzinger.

What is the problem?

- Works on Assume/Guarantee reasoning [Hoare, Gries, Lamport, Meyer, Back & von Wright] and Interfaces [deAlfaro-Henzinger 01] have well addressed the following central issues:
  - The principle of substituability and its consequences on the covariant/contravariant nature of refinement – assumptions must get weaker and guarantees stronger
  - How to compose contracts for different components
  - How to actually represent a contract in a computationally effective way, *e.g.,* the ***interface automata*** by deAlfaro-Henzinger.
- However, the concept of ***refining a set of contracts*** (attached to a same component) has not been considered so far, at least to our knowledge. This is, however, needed in order to address multiple viewpoints.

What is the problem?

- Works on Assume/Guarantee reasoning [Hoare, Gries, Lamport, Meyer, Back & von Wright] and Interfaces [deAlfaro-Henzinger 01] have well addressed the following central issues:
  - The principle of substituability and its consequences on the covariant/contravariant nature of refinement – assumptions must get weaker and guarantees stronger
  - How to compose contracts for different components
  - How to actually represent a contract in a computationally effective way, *e.g.,* the ***interface automata*** by deAlfaro-Henzinger.
- However, the concept of ***refining a set of contracts*** (attached to a same component) has not been considered so far, at least to our knowledge. This is, however, needed in order to address multiple viewpoints.
- In addition, requirements capture quite often consists in formulating a set of contracts attached to a component (even for the functional viewpoint alone); this is the way advocated by users in SPEEDS

Framework and Results

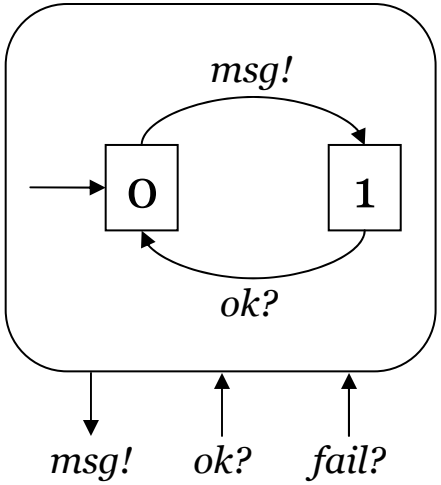
To help for clarity we consider the basic, “visual”, case of automata with associated product.

Of course a richer framework would be needed to account for heterogeneity due to multiple viewpoints

Candidate framework: interface automata
[deAlfaro-Henzinger 01]

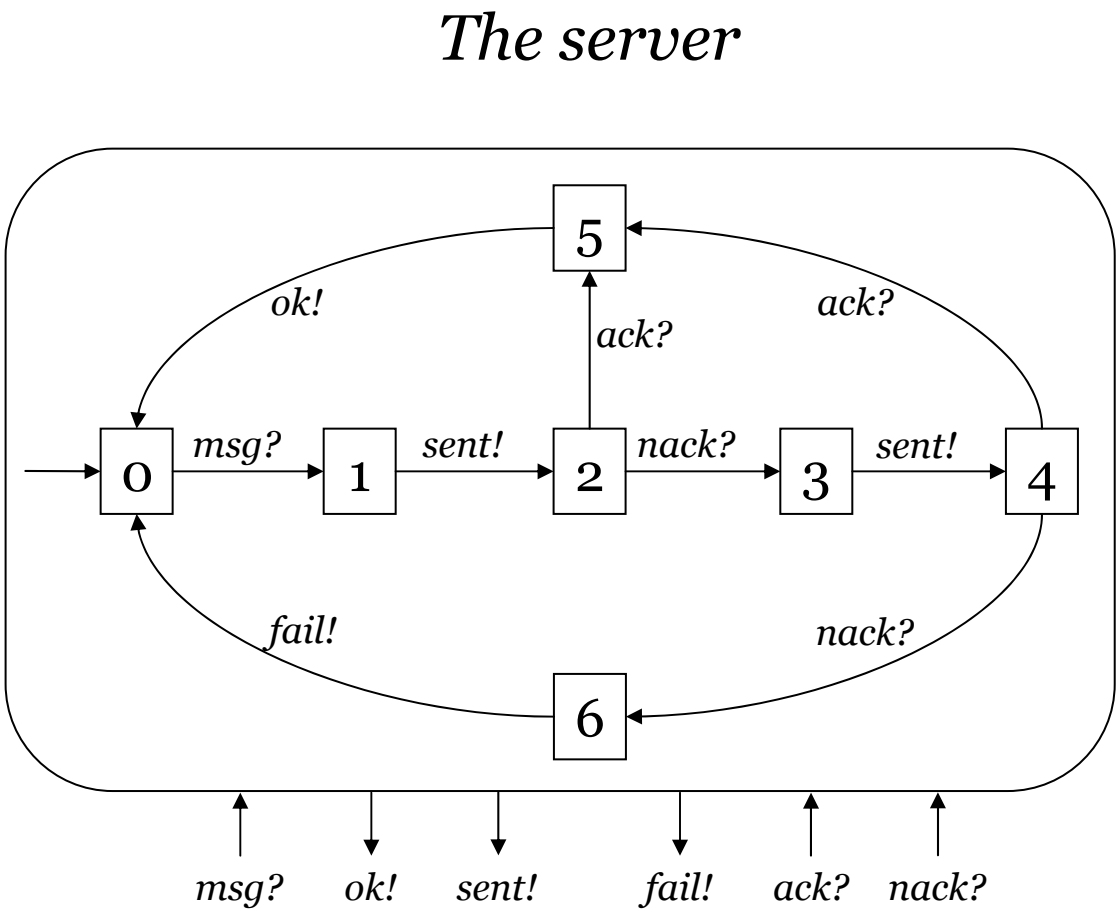
- i/o profile shown at the interface
- the automaton specifies who performs what in which state
- Assumption:
  - env. should never submit *fail*
  - when in state 1, env. can submit *ok*
- Guarantee: system responds as indicated

The client



Candidate framework: interface automata

[deAlfaro-Henzinger 01]

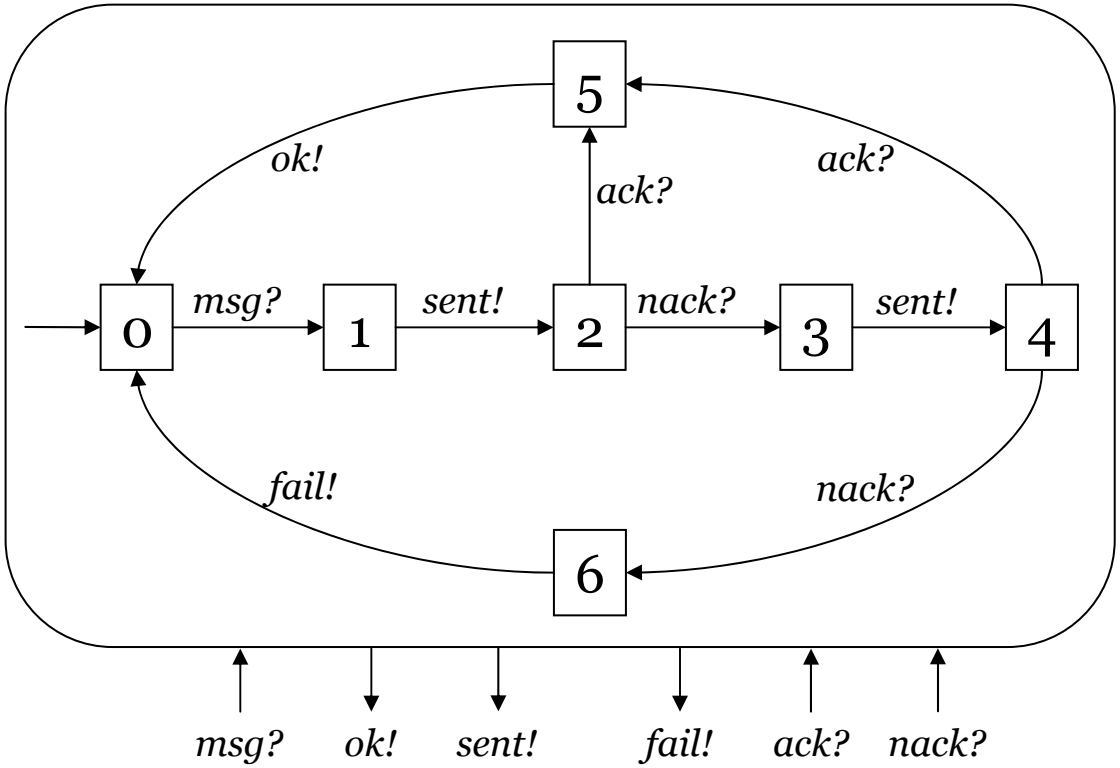


- i/o profile shown at the interface
- the automaton specifies who performs what in which state
- Assumption: environment offers what is allowed depending on the state
- Guarantee: system responds as indicated

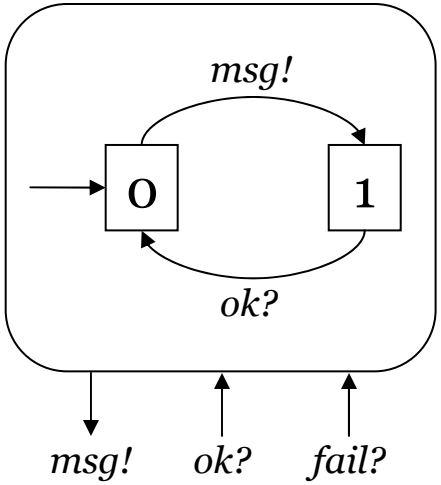
Candidate framework: interface automata

[deAlfaro-Henzinger 01]

The server

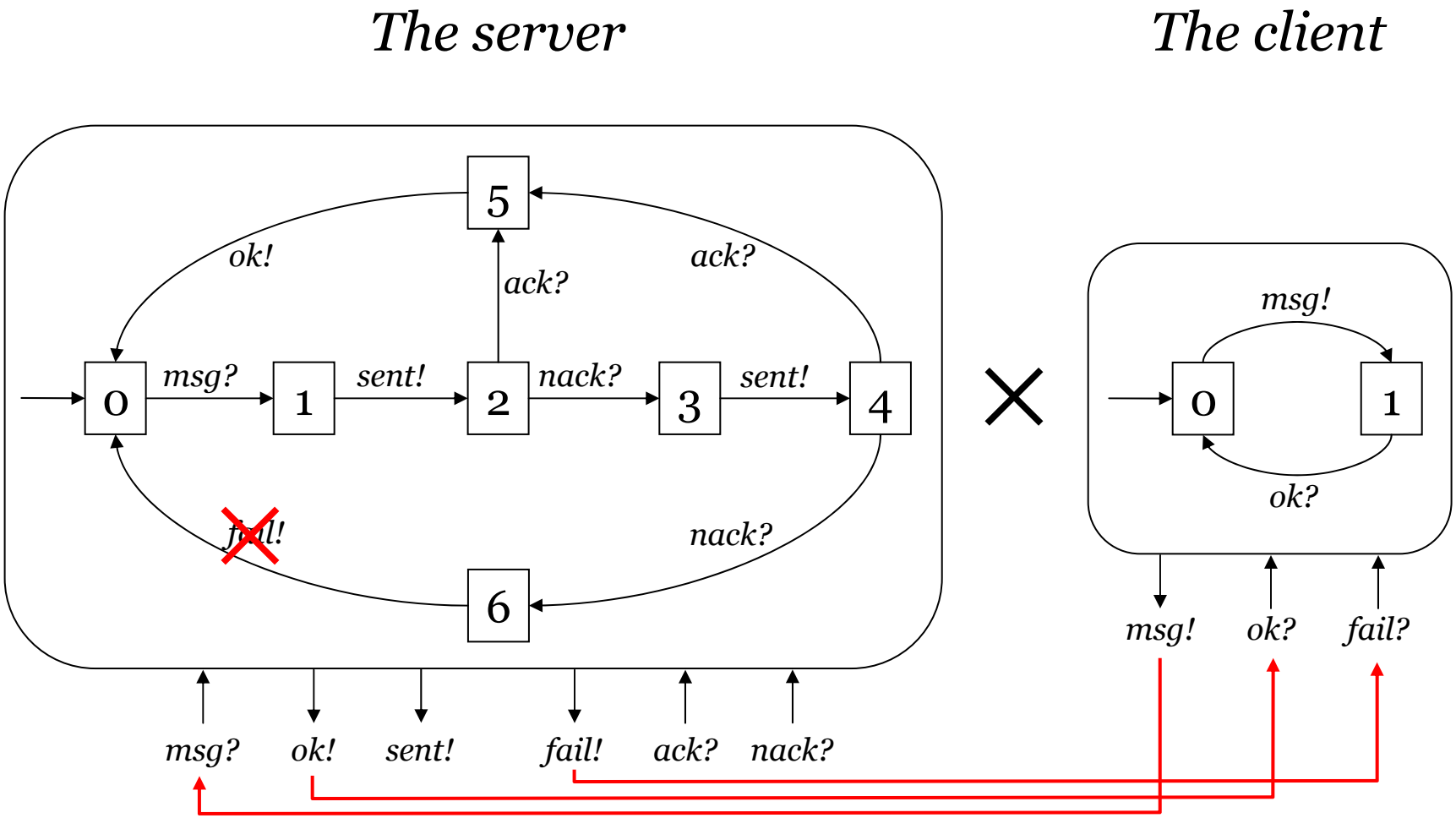


The client



Candidate framework: interface automata

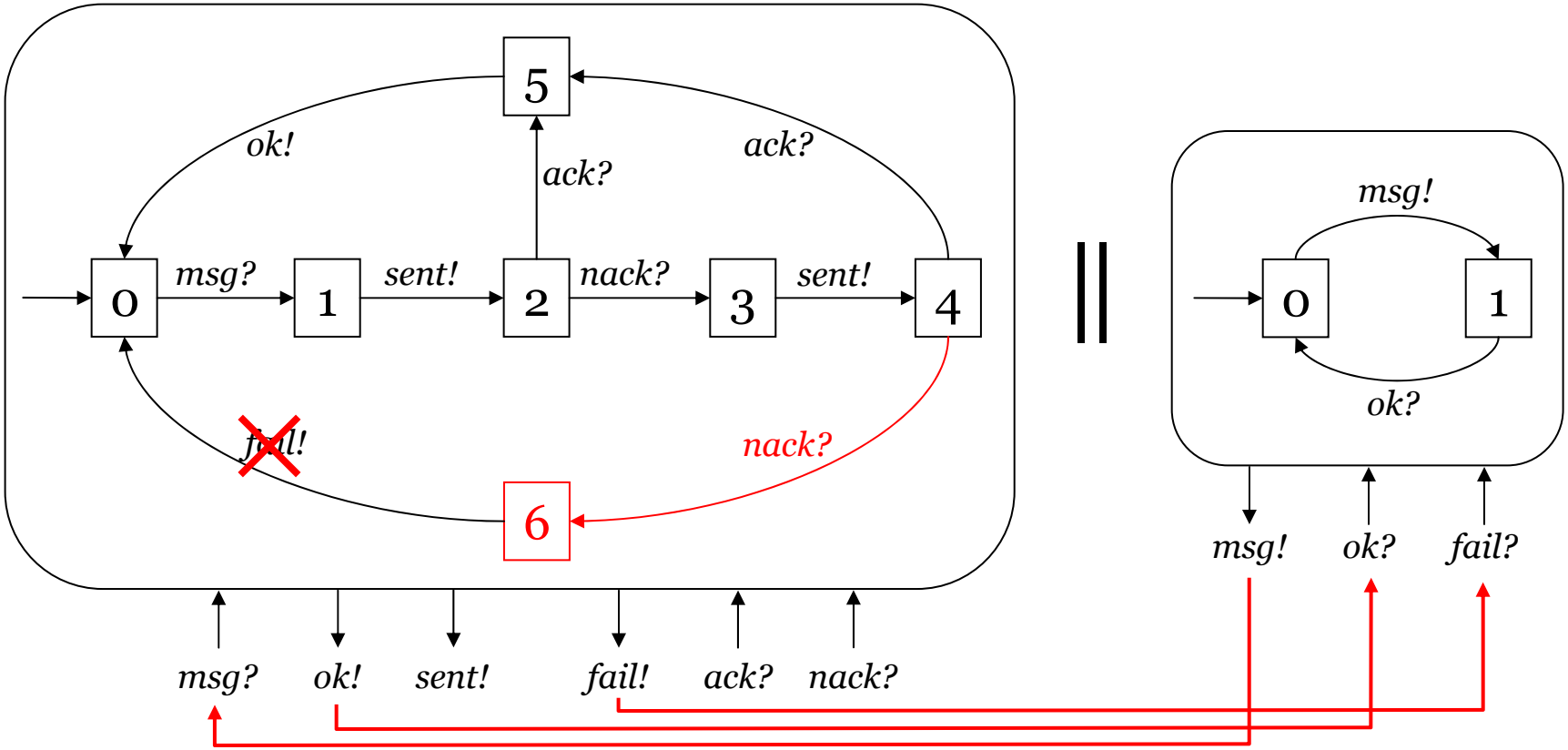
[deAlfaro-Henzinger 01]



Candidate framework: interface automata

[deAlfaro-Henzinger 01]

prune in order to avoid the system offering no response to the environment

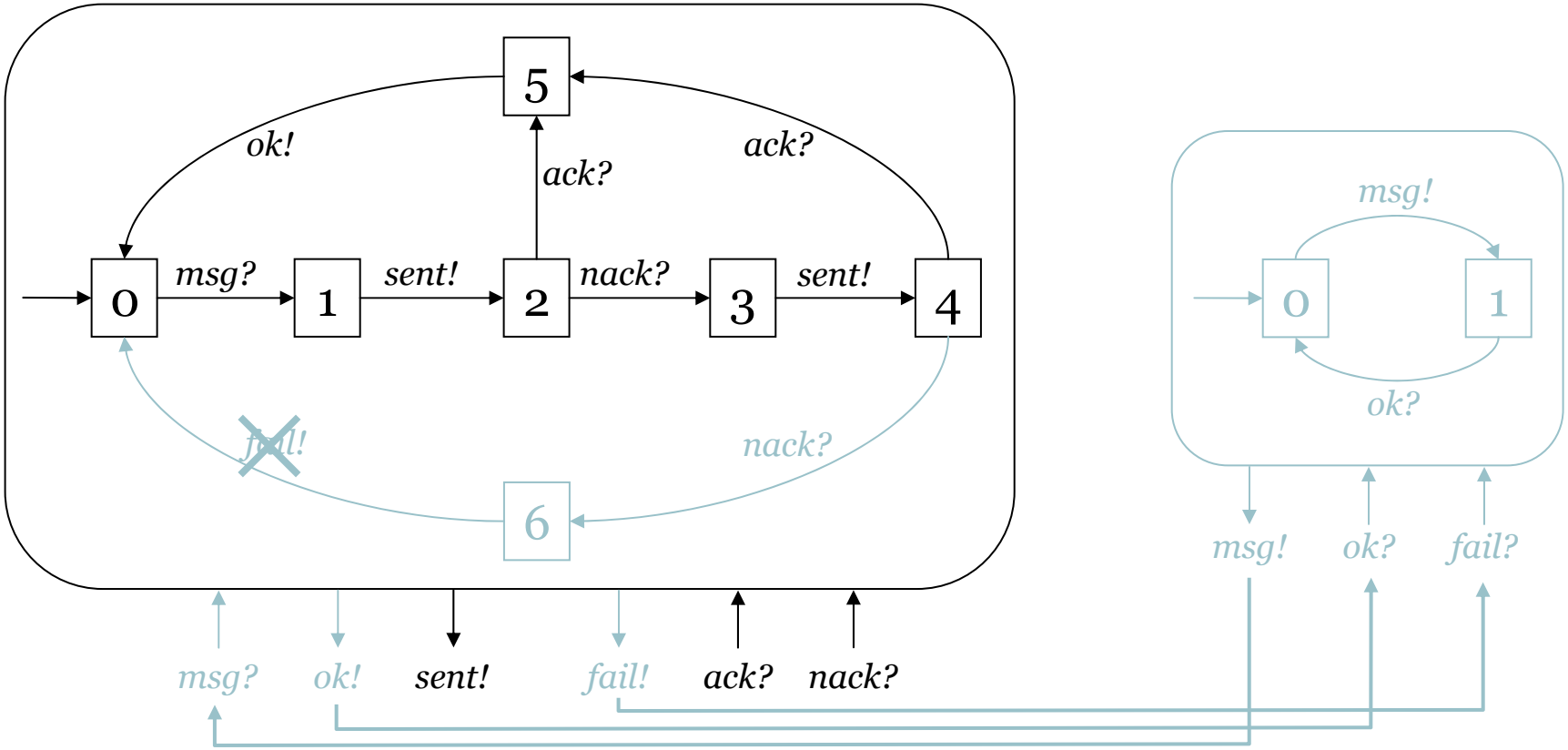


Candidate framework: interface automata

[deAlfaro-Henzinger 01]

the result tells

how the network should behave in order for the protocol to deliver service as expected by the client

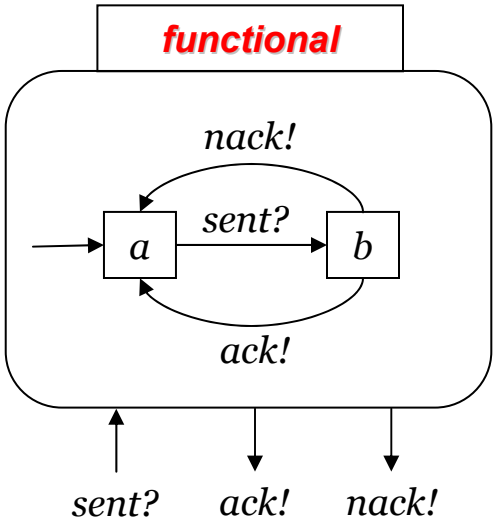


A refined story

- The *client* wants to see again the same story: $msg!ok?msg!ok?...$
- We have the same protocol component (the *server*), for reuse and possibly patch whenever needed
- In addition, we have some knowledge about the network:
  -
  -

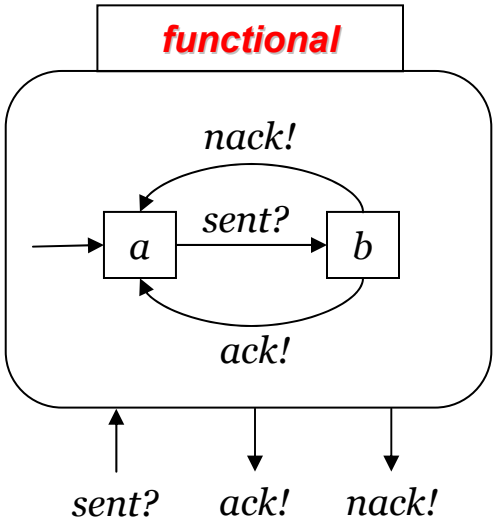
A refined story

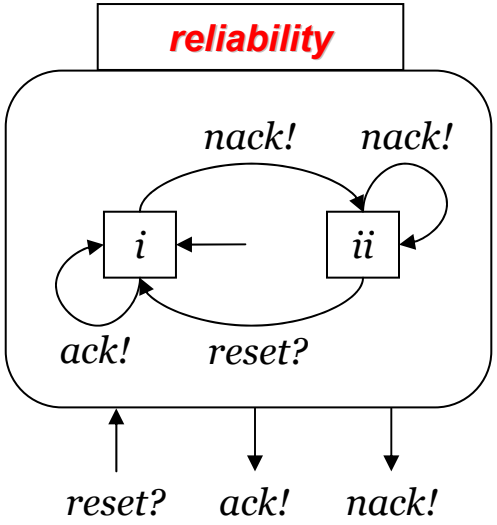
- The *client* wants to see again the same story: *msg!ok?msg!ok?*...
- We have the same protocol component (the *server*), for reuse and possibly patch whenever needed
- In addition, we have some knowledge about the network:
  - ***functional***: when receiving a request *sent* from the server, it answers either *ack* (in case transmission succeeds) or *nack* (otherwise)
  -



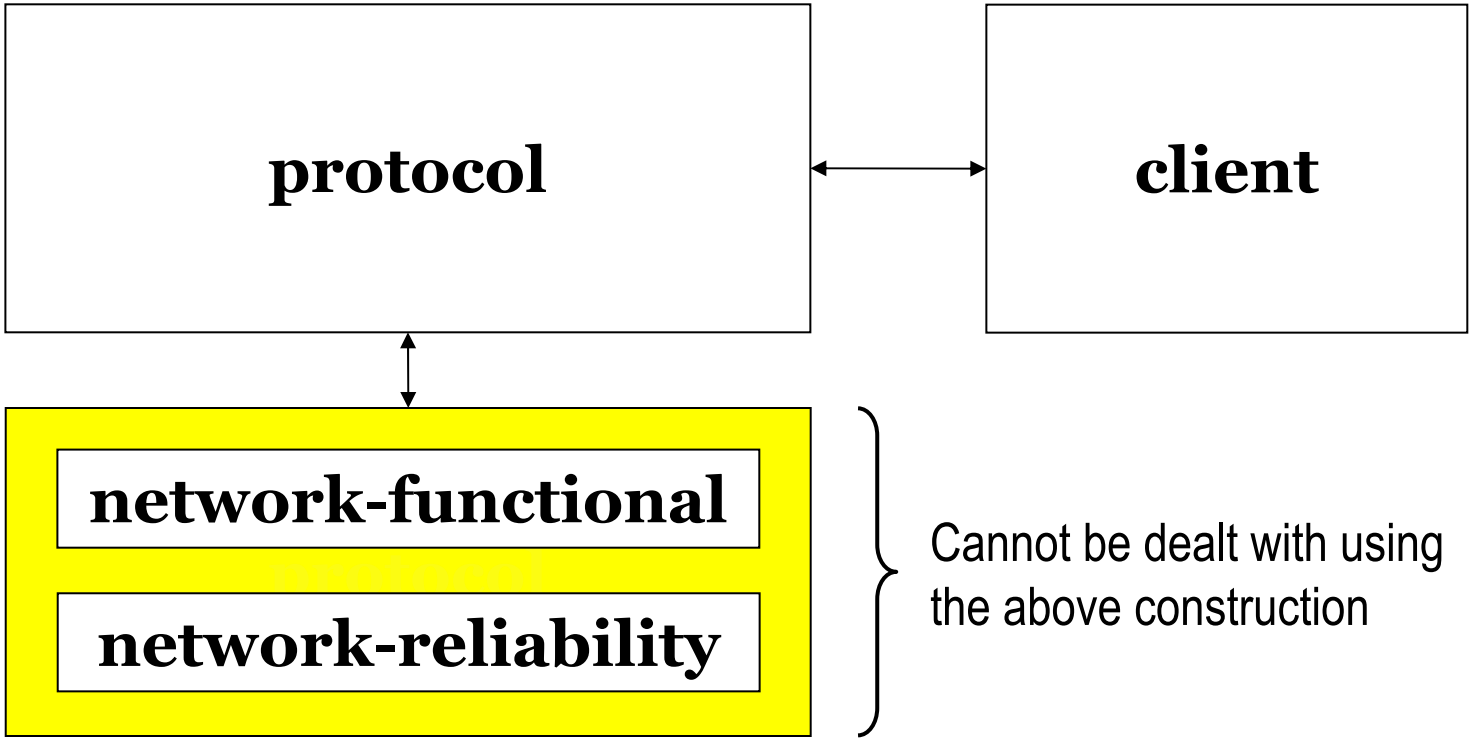
A refined story

- The *client* wants to see again the same story: *msg!ok?msg!ok?*...
- We have the same protocol component (the *server*), for reuse and possibly patch whenever needed
- In addition, we have some knowledge about the network:
  - ***functional***: when receiving a request *sent* from the server, it can answer either *ack* (in case transmission succeeds) or *nack* (otherwise)
  - ***reliability***: the network will not stop failing to transmit until a *reset* action is performed by the supervisor





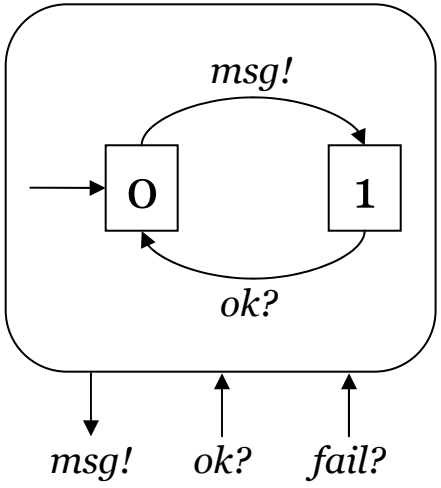
The corresponding architecture: multiple viewpoint



The above difficulty can be turned around by using a simple trick. It mostly consists in revisiting the interpretation of *Interface Automata* as a pair {Assumption,Guarantee}. Call for a while the solution:

A/G-automata

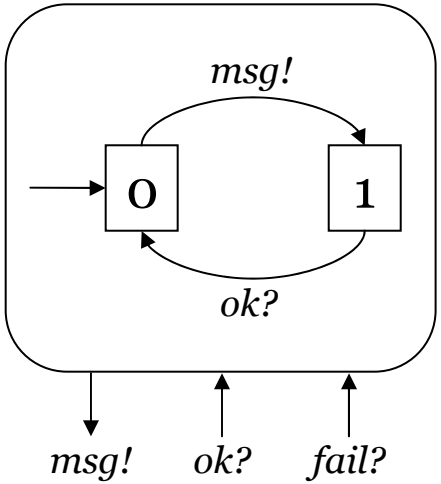
A/G-automaton for the client

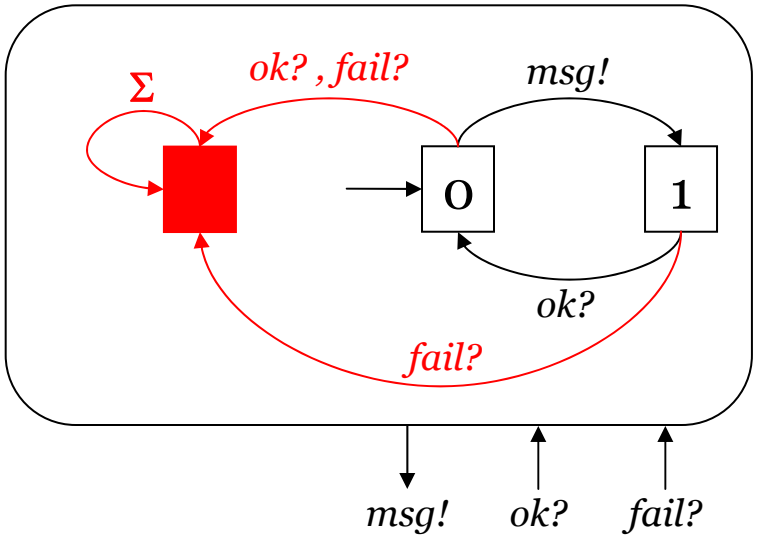


- i/o profile shown at the interface
- the automaton specifies who can perform what in which state
- Assumption:
  - env. should never submit *fail*
  - when in state 1, env. can submit *ok*
- Guarantee: system responds as indicated

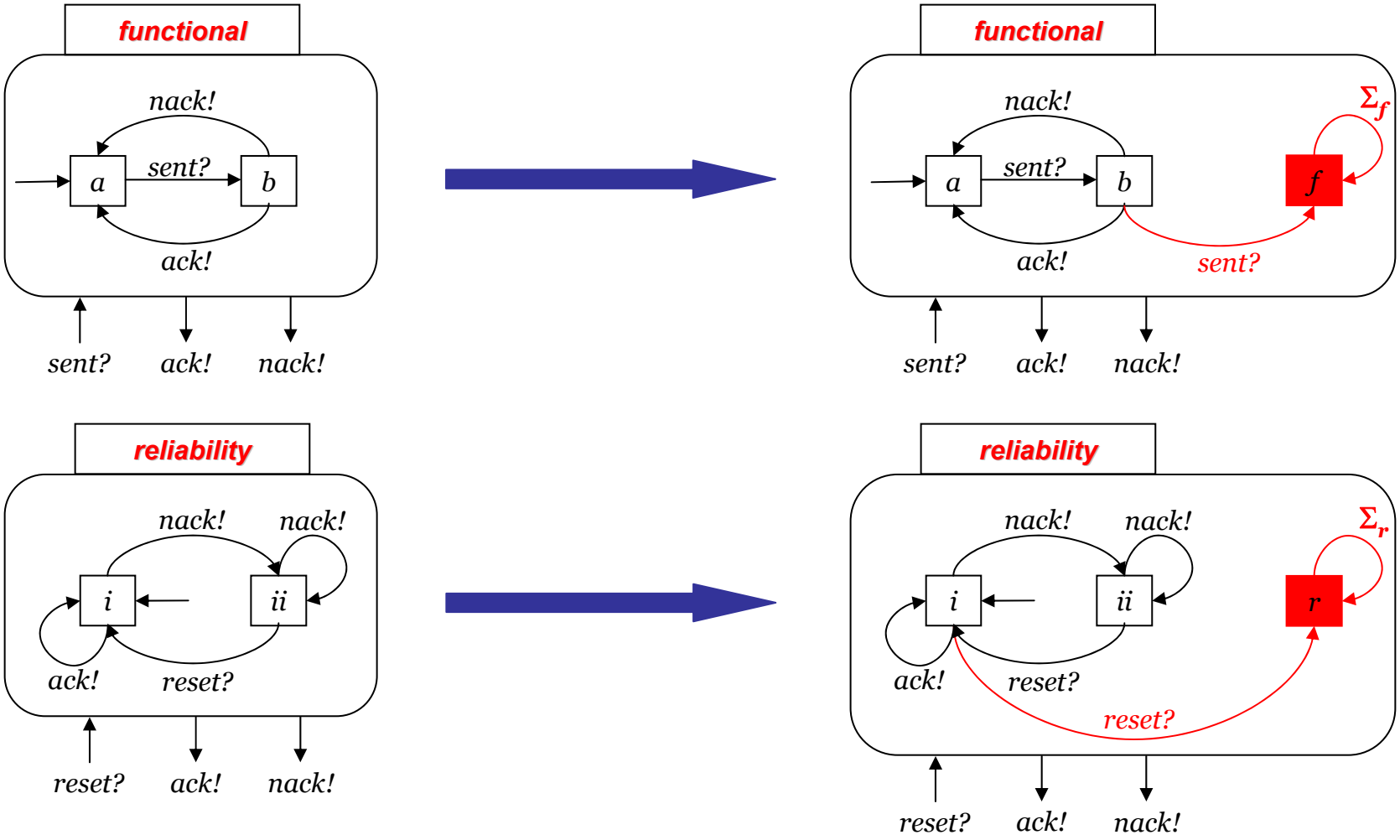
A/G-automaton for the client

- i/o profile shown at the interface
- the automaton specifies who can perform what in which state
- Assumption:
  - env. should never submit *fail*
  - when in state 1, env. can submit *ok*
- Guarantee: system responds as indicated
- Interpret the above interface automaton as shown here:
  - Input enabled
  - Moves to trap when assumption gets violated
  - See ?/! as side labels

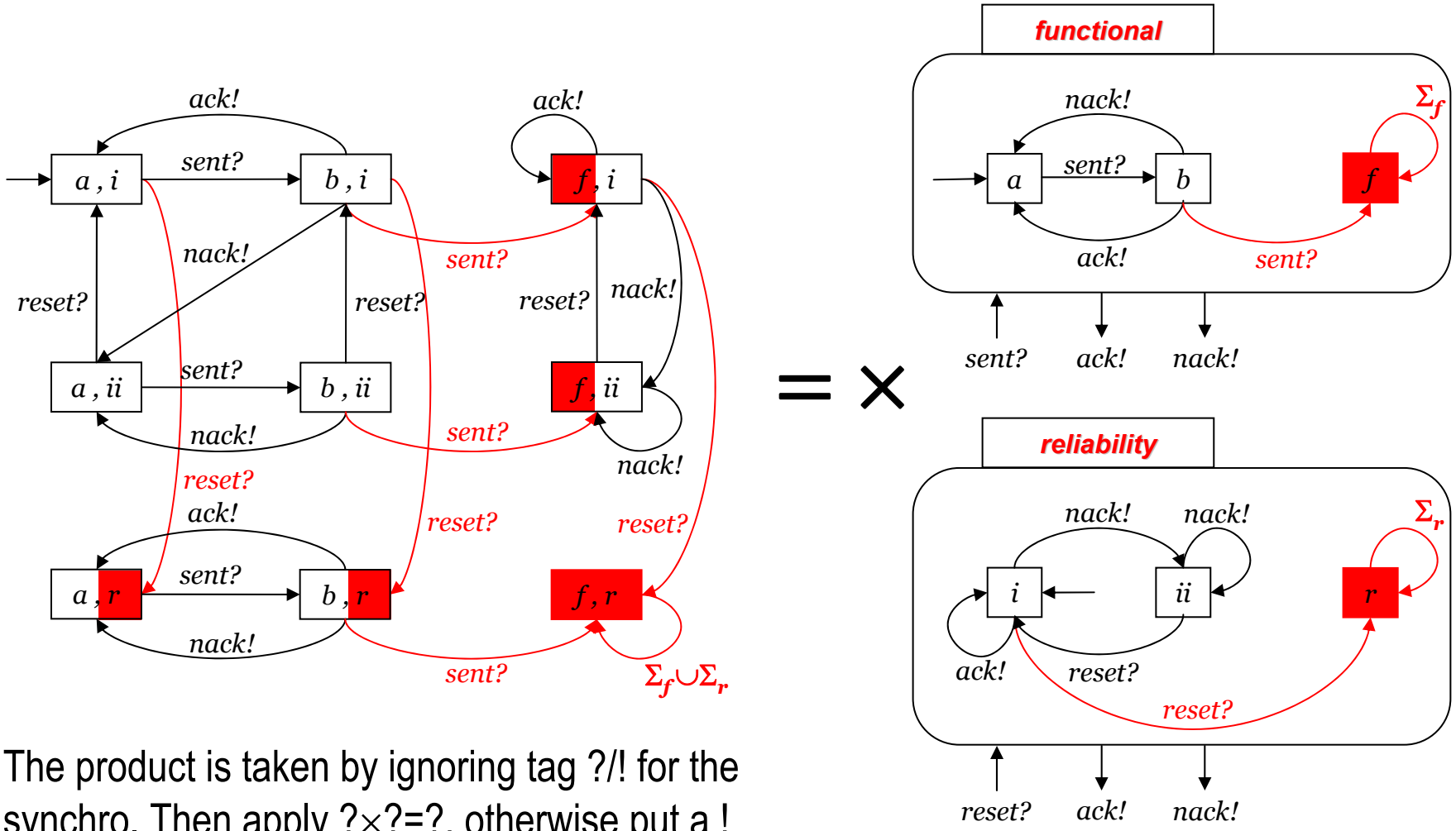




A/G-automaton for the network



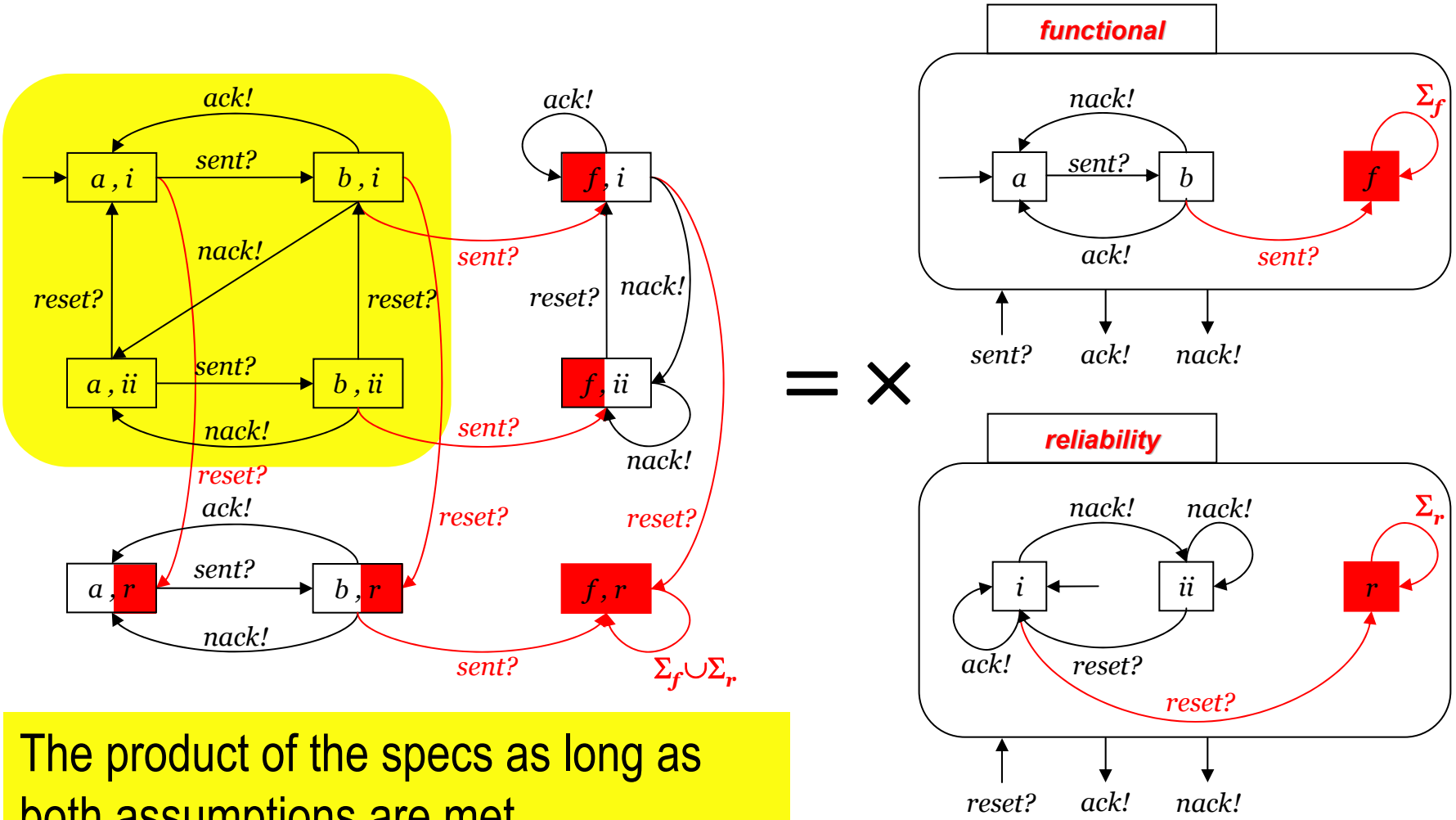
A/G-automaton for the network



The product is taken by ignoring tag ?/! for the synchro. Then apply ?×?=?, otherwise put a !

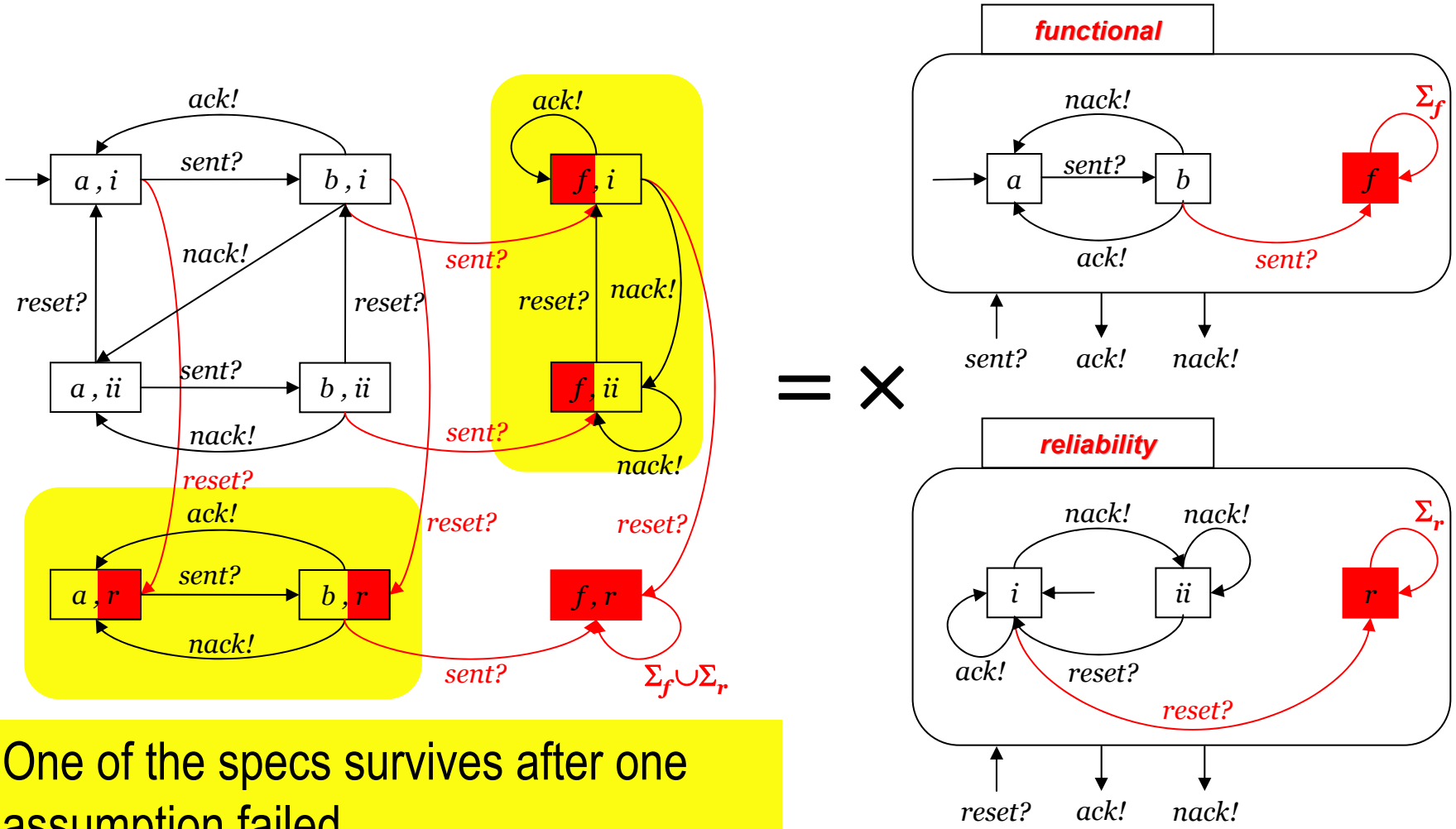
(A few self-loops were ignored)

A/G-automaton for the network



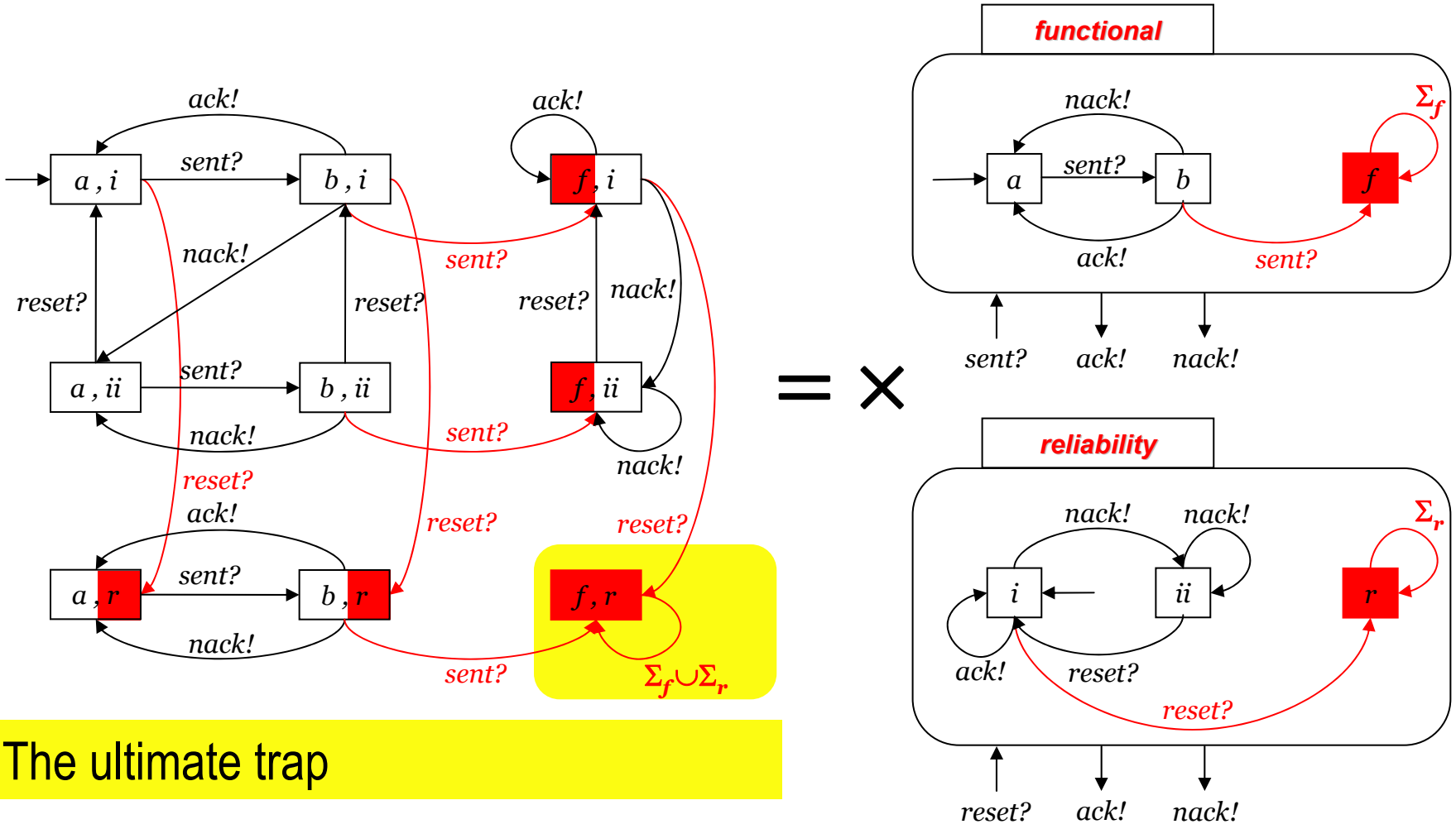
The product of the specs as long as both assumptions are met

A/G-automaton for the network



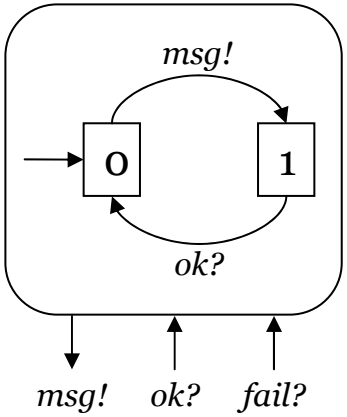
One of the specs survives after one assumption failed

A/G-automaton for the network



The ultimate trap

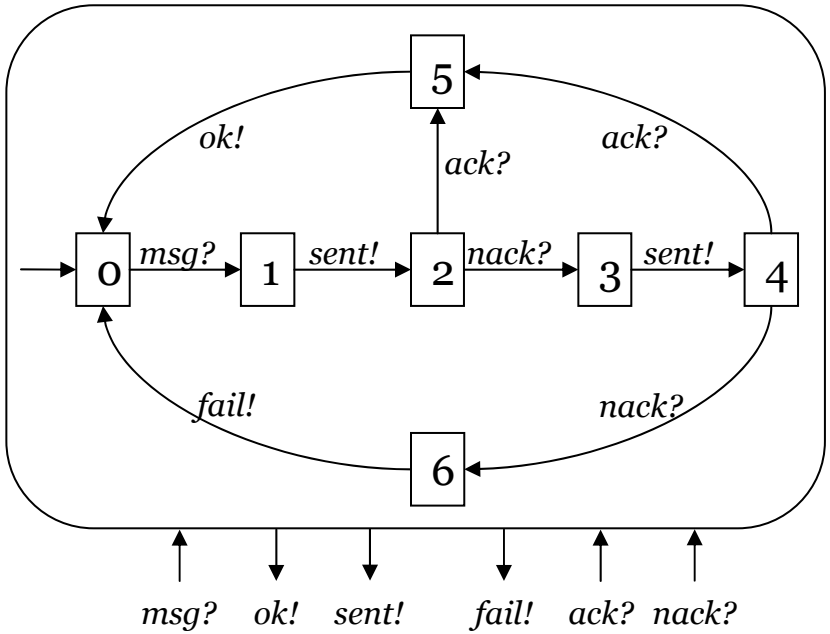
A/G-automaton for the client-server



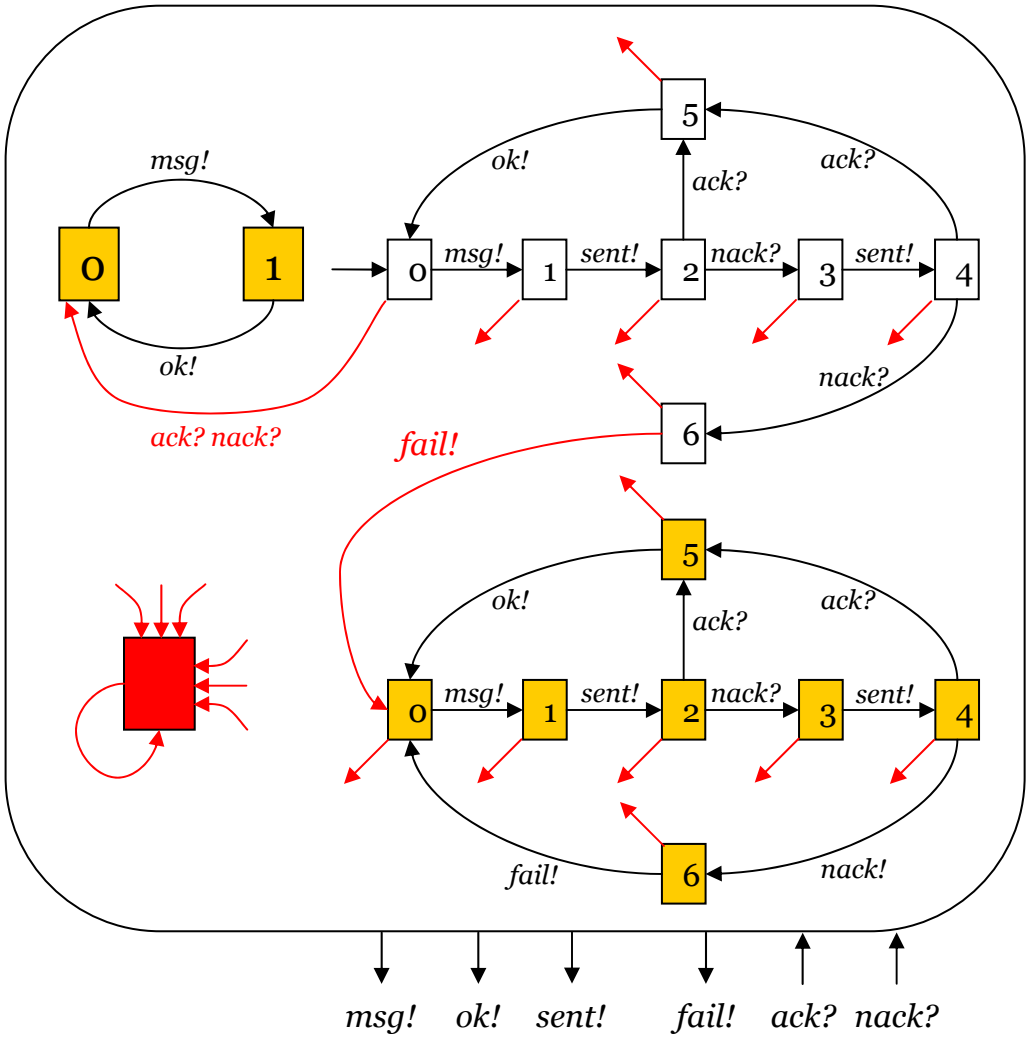
$\times_{AG}$ defined as

1. AG-complete
2. Take product

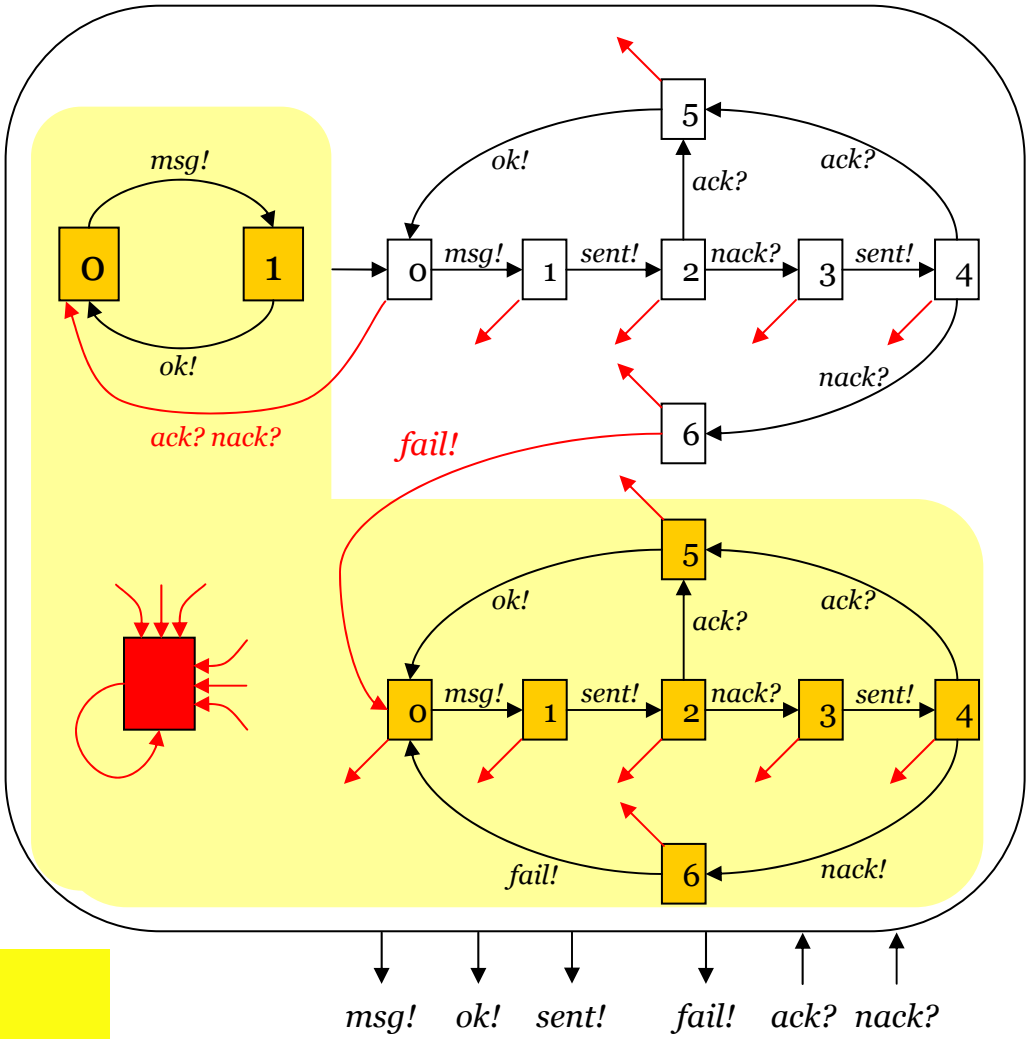
$= \times_{AG}$



A/G-automaton for the client-server

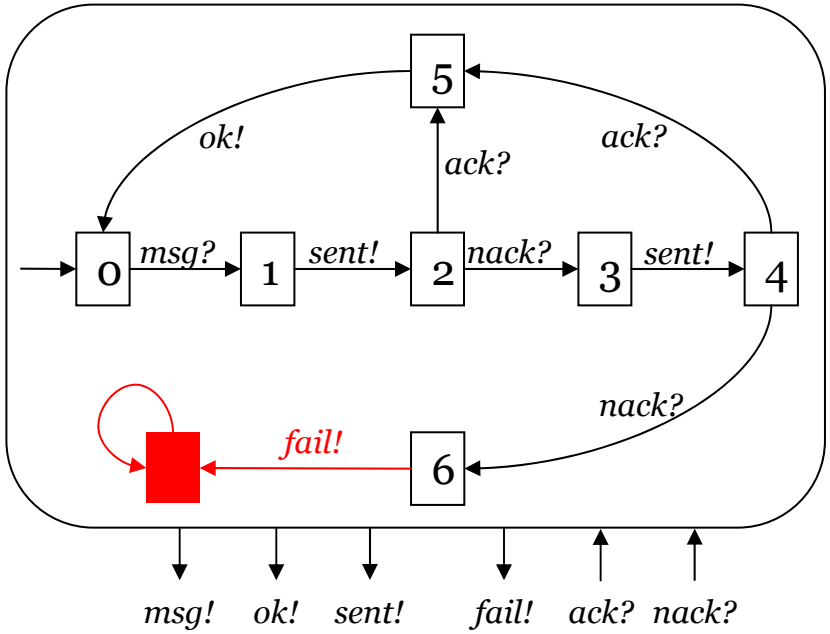


A/G-automaton for the client-server

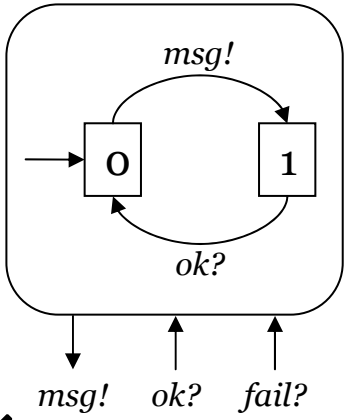


If this product is to be seen as a single component, then the detail of how assumptions fail may be hidden to the environment and collapsed to a single trap

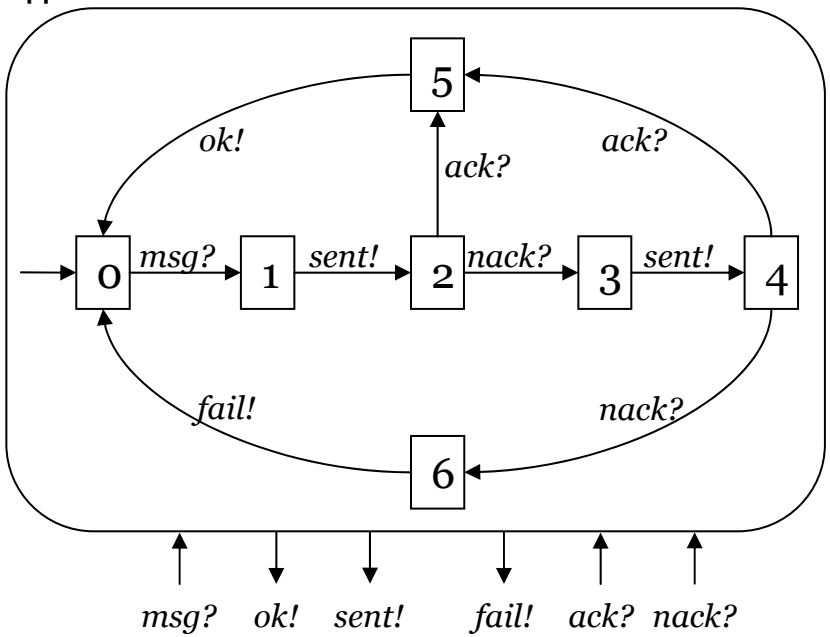
A/G-automaton for the client-server



=



$\times_{||}$

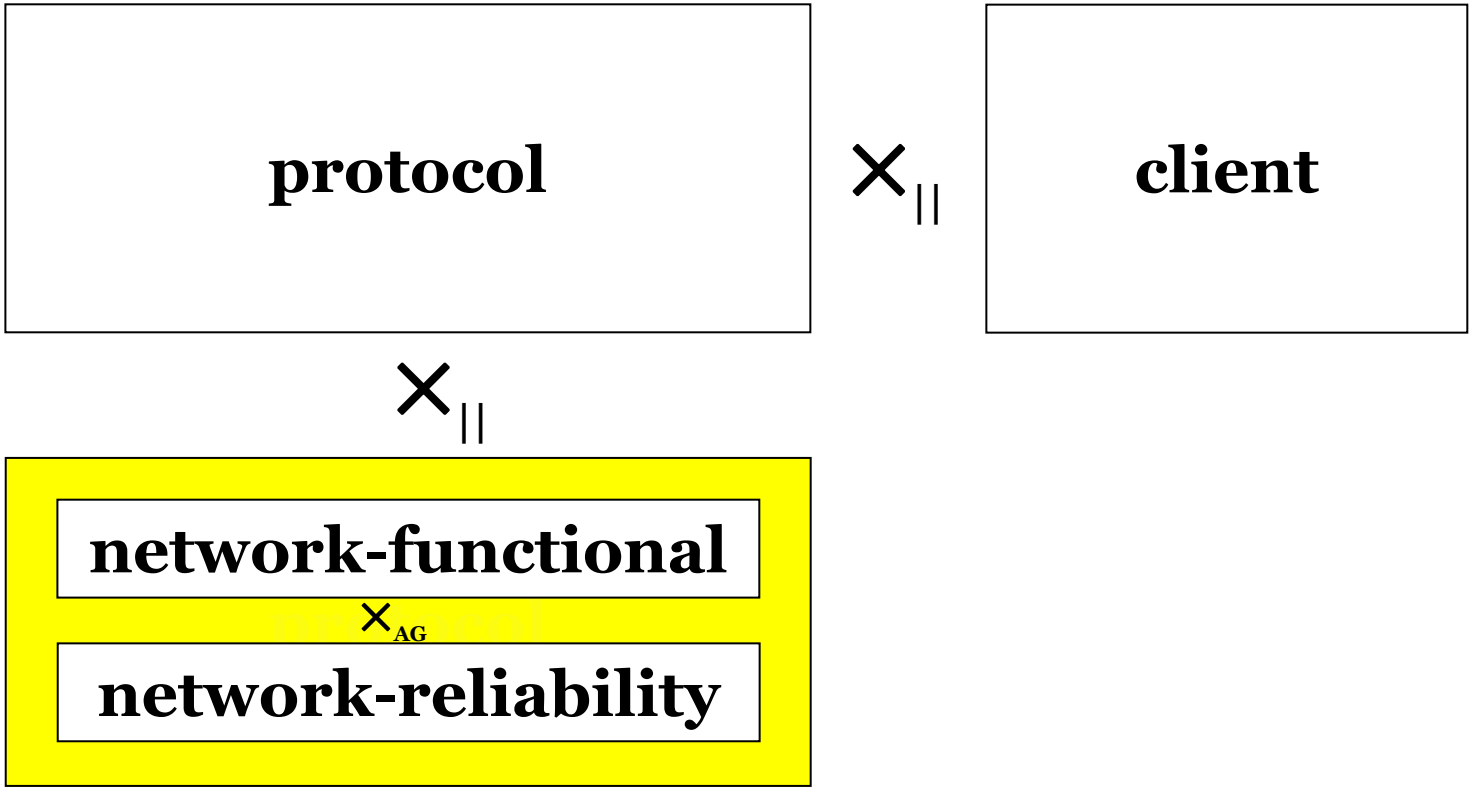


$\times_{||}$ defined as

1. AG-complete
2. Take product
3. Collapse traps

If this product is to be seen as a single component, then the detail of how assumptions fail may be hidden to the environment and collapsed to a single trap

Multiple viewpoint contract for a system architecture?



This seems nice but looks like an ah-doc solution. How can we formalize this?

Contracts as Modal Automata

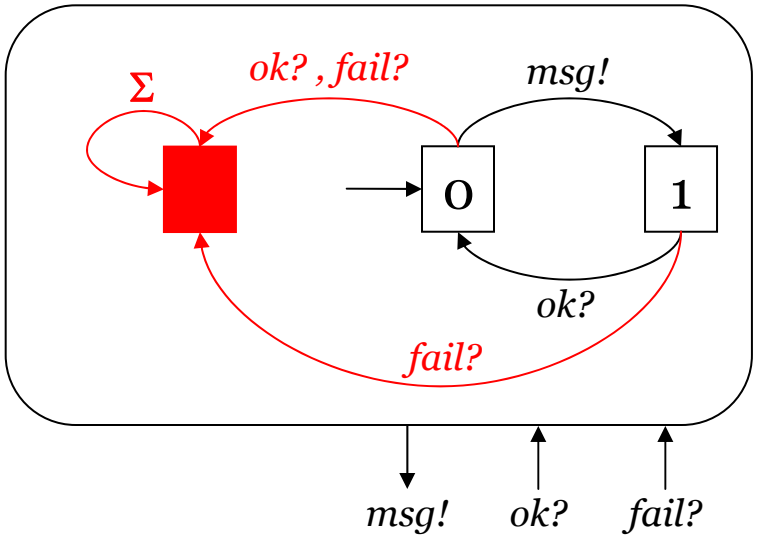
Modal Automata

[Kim Larsen 1989, Kim Larsen &al. 2007]

- Modal Automata are automata with *may* and *must* transitions
- They were proposed by Kim Larsen in 1989 to study refinement specification and further compared to Interface Automata in 2007 in the context of product lines

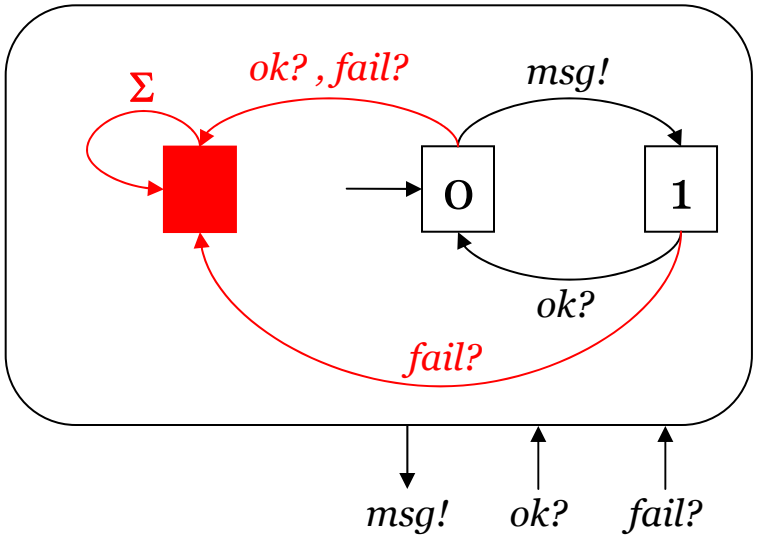
Introducing Modal Automata

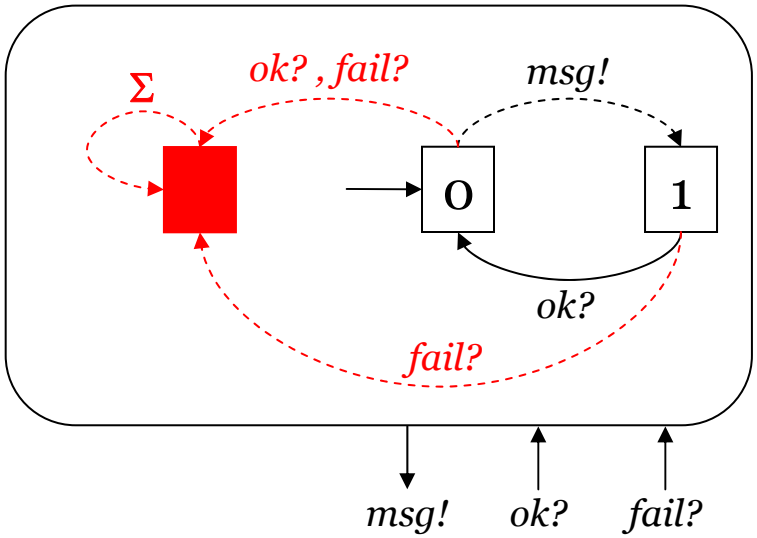
- Top diagram:
  - Nominal behavior in black
  - Degraded behavior in red
  - Input enabled



Introducing Modal Automata

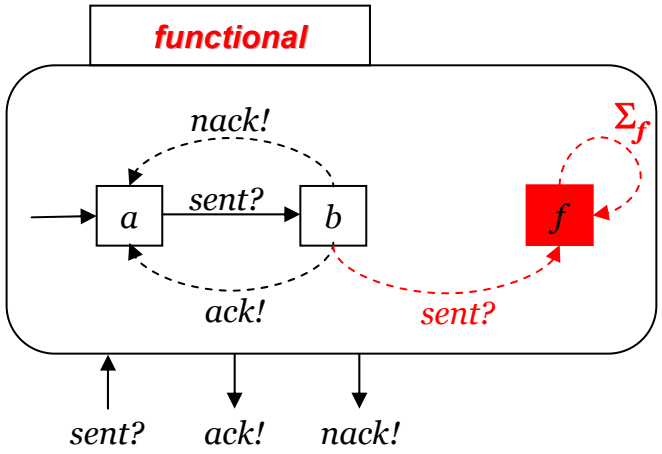
- Top diagram:
  - Nominal behavior in black
  - Degraded behavior in red
  - Input enabled
- Bottom diagram:
  - Keep colors as before, useful for interpreting nominal/degraded modes
  - Use modal automata with *may/must* transitions
    - Inputs in nominal behavior are *must* (input enabled)
    - Outputs in nominal behavior are *may* (best effort)
    - Degraded mode is *may*

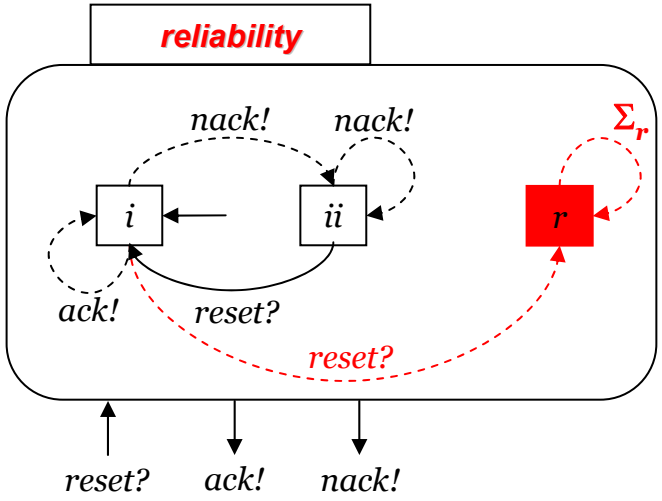




Introducing Modal Automata

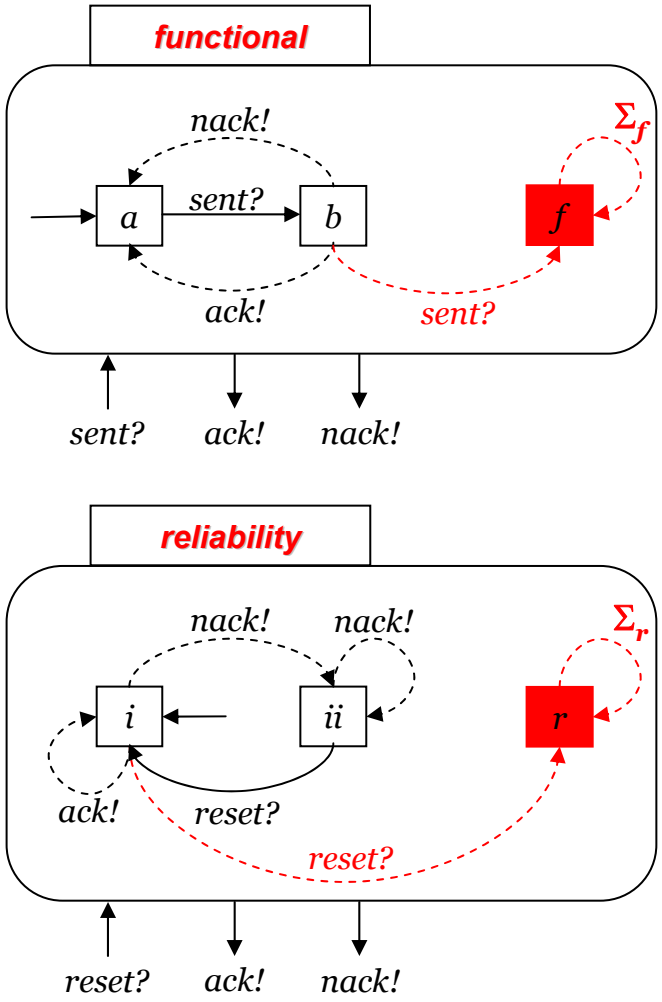
1. **S ::** $\forall$**q:** $must(q) \subseteq may(q)$
2. **C** $\models$ **S ::** $\forall$**q:**
   - $must(q) \subseteq C(q) \subseteq may(q)$
3. **S** $\leq$ **S' ::** $\forall$**q~q':**
   - $must(q) \supseteq must'(q')$
   - $may(q) \subseteq may'(q')$
4. **S**$\wedge$**S'**
   - $must(q) \cup must'(q')$
   - $may(q) \cap may'(q')$
   - and then clean the result to ensure 1.
5. **S**$\otimes$**S'**
   - $must(q) \cap must'(q')$
   - $may(q) \cap may'(q')$



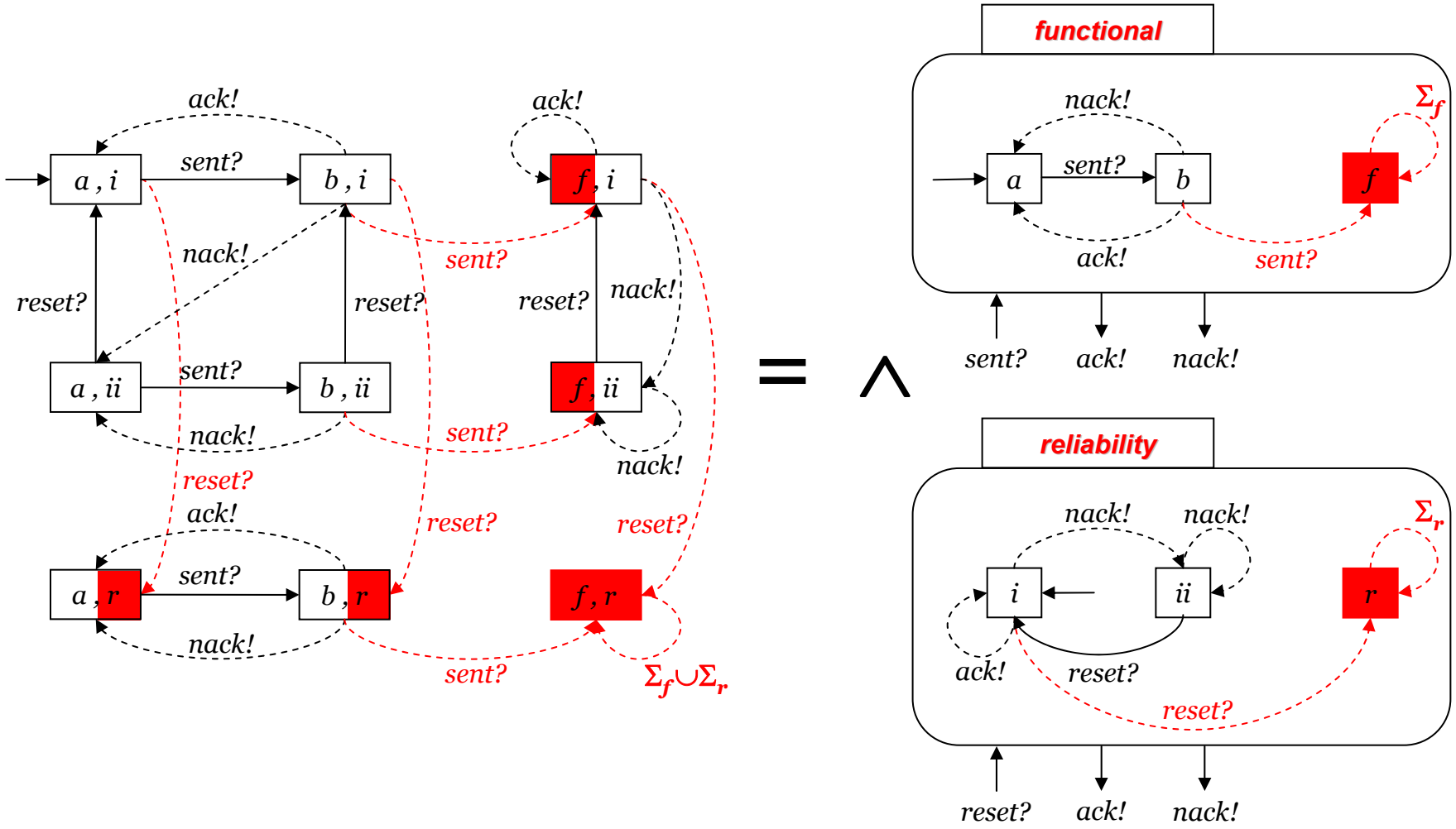


Supporting substituability for multiple viewpoint contracts

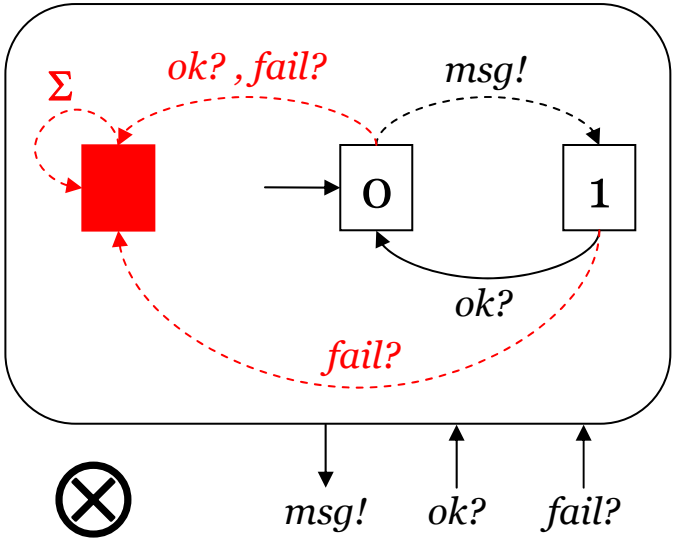
- **S ::** $\forall$**q:** $must(\mathbf{q}) \subseteq may(\mathbf{q})$
- **C** $\models$ **S ::** $\forall$**q:**
  - $must(\mathbf{q}) \subseteq \mathbf{C}(\mathbf{q}) \subseteq may(\mathbf{q})$
- **S** $\leq$ **S' ::** $\forall$**q~q':**
  - $must(\mathbf{q}) \supseteq must'(\mathbf{q}')$
  - $may(\mathbf{q}) \subseteq may'(\mathbf{q}')$
- **S**$\wedge$**S'**
  - $must(\mathbf{q}) \cup must'(\mathbf{q}')$
  - $may(\mathbf{q}) \cap may'(\mathbf{q}')$
  - and then clean the result to ensure 1.
- **C** $\models$ **S** and **C** $\models$ **S'** iff **C** $\models$ **S** $\wedge$**S'**



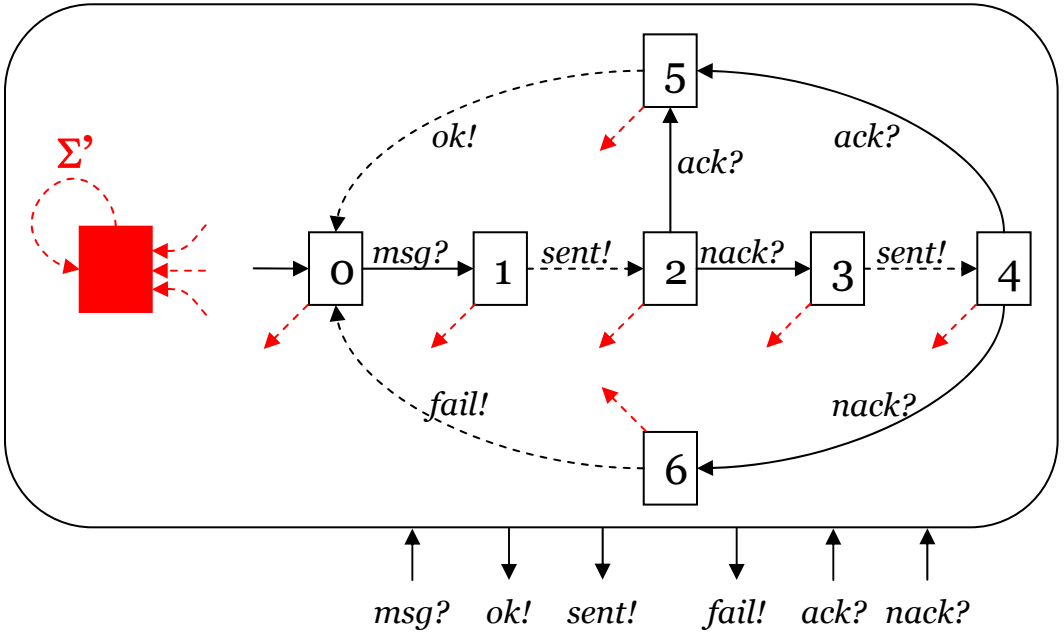
Supporting substituability for multiple viewpoint contracts



Supporting substituability for multiple components

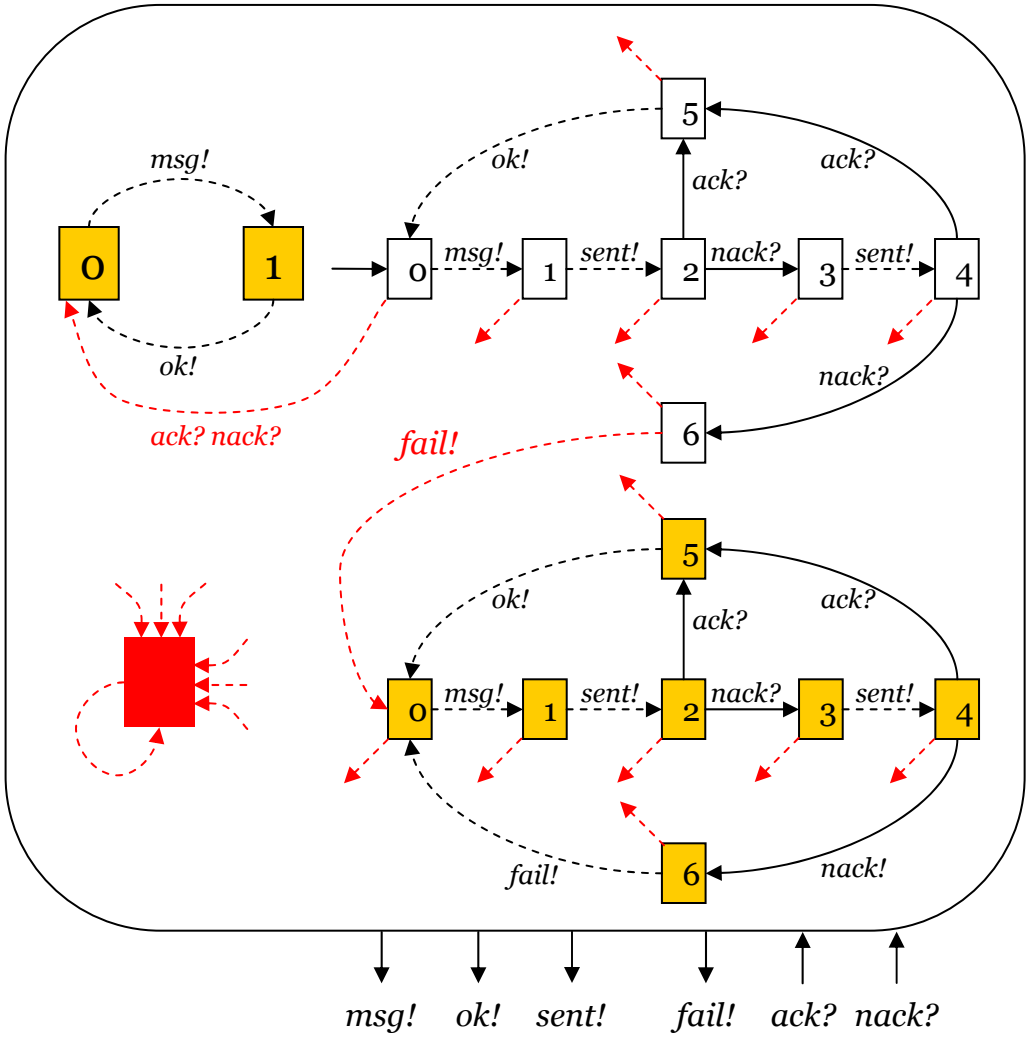


- **S⊗S'**
  - *must*(**q**) ∩ *must'*(**q'**)
  - *may*(**q**) ∩ *may'*(**q'**)
- **C** ⊨ **S** and **C'** ⊨ **S'** implies **C×C'** ⊨ **S⊗S'**



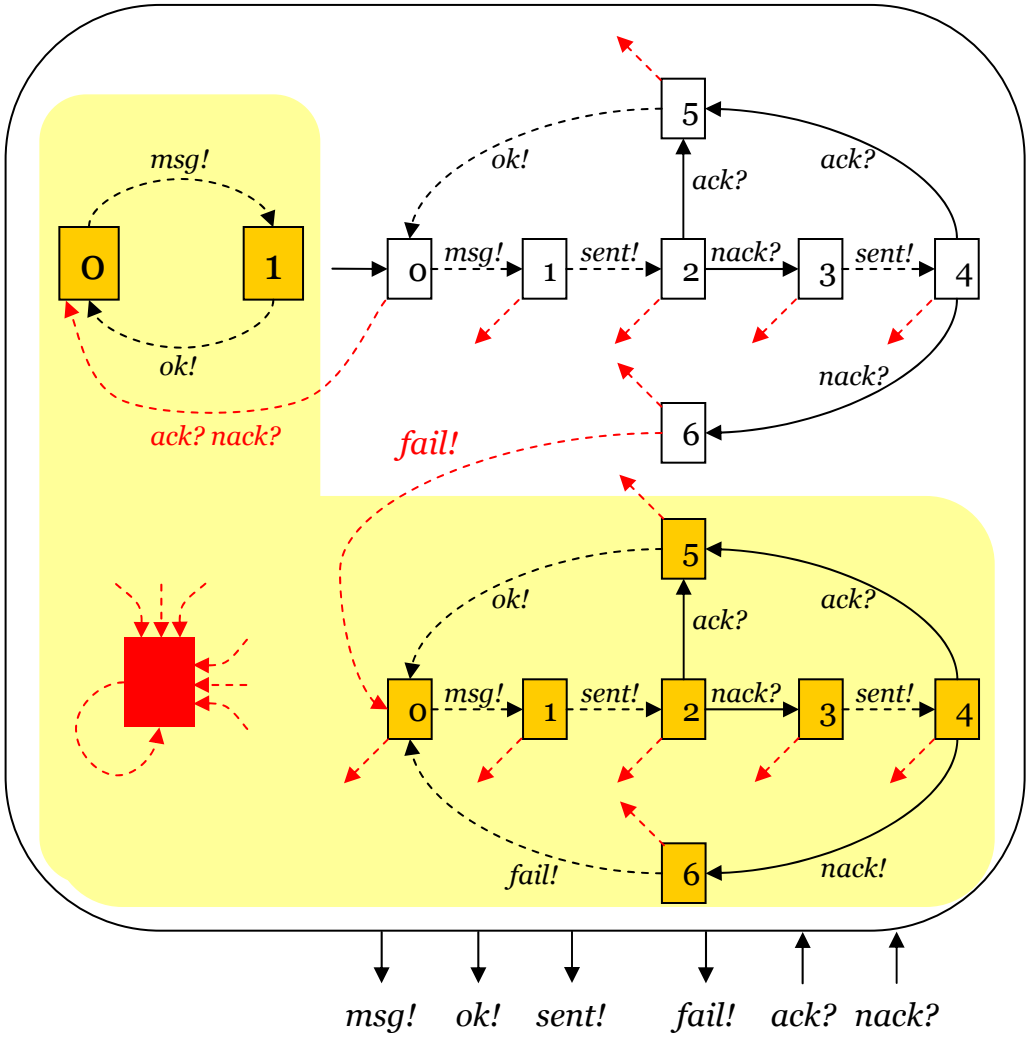
Supporting substituability for multiple components

- **S⊗S'**
  - *must*(**q**) ∩ *must'*(**q'**)
  - *may*(**q**) ∩ *may'*(**q'**)
- **C** ⊨ **S** and **C'** ⊨ **S'** implies **C×C'** ⊨ **S⊗S'**

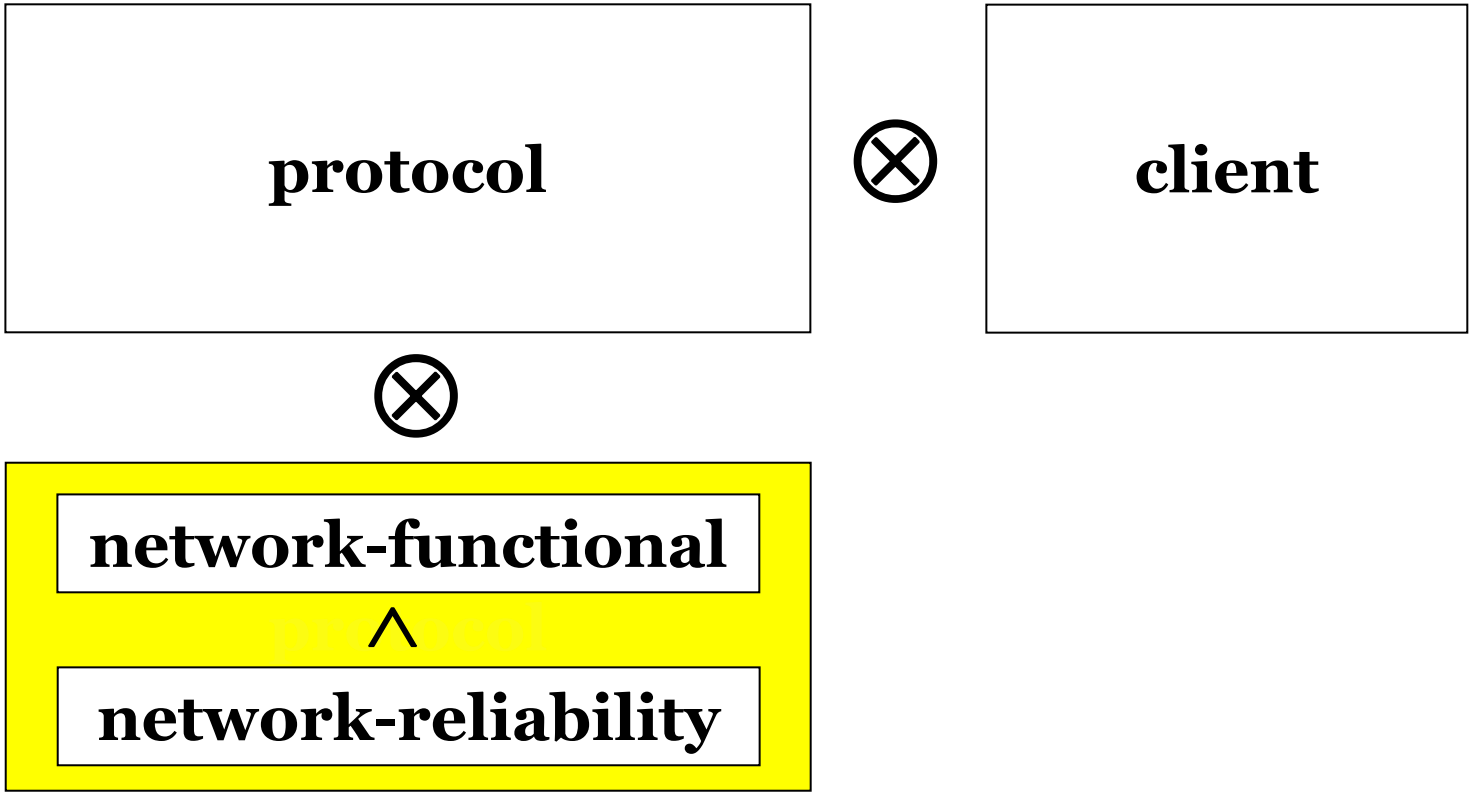


Supporting substituability for multiple components

- **S⊗S'**
  - *must*(**q**) ∩ *must'*(**q'**)
  - *may*(**q**) ∩ *may'*(**q'**)
- **C** ⊨ **S** and **C'** ⊨ **S'** implies **C×C'** ⊨ **S⊗S'**
- Collapse trap + semi-traps to a single abstract trap? (not part of contract semantics)



Multiple viewpoint contract for a system architecture



The designer can either

1. design components based on all viewpoints and then
2. assemble components,

or perform the converse (or a mix).

Can we relate the two?

Recall: Embedded systems possess many components + different viewpoints

- Function
- Timing
- Reliability
- Energy
- QoS
- …

- The designer may want to:
  - consider all viewpoints for each component
  - implement each component
  - compose the implementations

- Alternatively she may want to consider viewpoints incrementally:
  - consider all viewpoints for each component except Safety + QoS
  - implement each component
  - compose the implementations
  - Revisit her design for safety and QoS, possibly with a different, coarser grain, architecture

A fundamental theorem

$$\mathbf{S \otimes (S' \wedge S'') \leq (S \otimes S') \wedge (S \otimes S'')}$$

Component centric Vewpoint centric

Viewpoint Centric design
leaves more room for implementations than
Component Centric design

So far our contracts are not split into assumptions and guarantees. Can we do this?

Contracts as pairs {Assume,Guarantee}
C=G/A

Contract = Guarantee given Assumption

Idea: Residuation

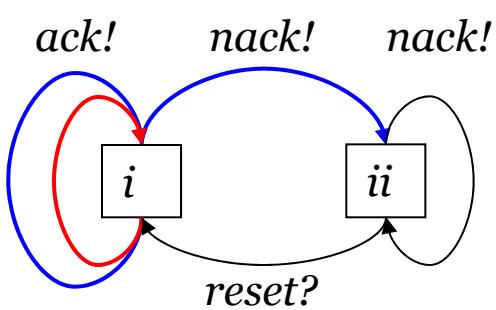
- Want to interpret our contracts **C** as "under assumption **A**, then **G** must be guaranteed"
- Equivalently: having the environment offering **A** to me, what should I do to ensure **G**?
- This is a "patch" synthesis problem. Modal automata offer a nice framework for this: contract **C** is solution of $\mathbf{A} \otimes ? = \mathbf{G}$
- But map $\mathbf{X} \rightarrow \mathbf{A} \otimes \mathbf{X}$ is not bijective, so the above equation has no simple solution.
- ***Residuation theory solves the problem.***

Generalizing Modal Automata to Acceptance Automata [J-B. Raclet 2007]

- Modal Automata are automata with *may* and *must* transitions
- They were proposed by Kim Larsen in 1989 to study refinement specification and further compared to Interface Automata in 2007
- J-B Raclet has seen modal automata as specifications **S**. He has studied ***residuation*** **S/M**.
- He has proposed a novel, mathematically simpler and more powerful, framework: ***Acceptance Specifications***. Roughly speaking, Acceptance Specifications are sets of automata with suitable operations.

Acceptance specifications (subsume Modal Automata)

- Acceptance specification: $\mathbf{S=(\Sigma,Acc)}$
  - Σ : alphabet
  - $Acc: \Sigma^* \rightarrow \wp(\wp(\Sigma))$ is a map, associating, to each finite word $\mathbf{u}$, a set of sets of possible transitions to continue $\mathbf{u}$ with
  - We require that, for finite word $\mathbf{u}$: $Acc(\mathbf{u}) \neq \emptyset$
  - Alphabet Equalization: $\Sigma \subset \Sigma'$ $\quad Acc(\mathbf{u}) \rightarrow \{ \mathbf{X} \cup (\Sigma' \setminus \Sigma) \mid \mathbf{X} \in Acc(\mathbf{u}) \}$



$Acc(\mathrm{i}) = \{\{ack\},\{ack,nack\}\}$
$Acc(\mathrm{ii}) = \{\{nack,reset\}\}$

Acceptance specifications (subsume Modal Automata)

- Acceptance specification: $\mathbf{S=(\Sigma,Acc)}$
  - Σ : alphabet
  - $Acc: \Sigma^* \rightarrow \wp(\wp(\Sigma))$ is a map, associating, to each finite word $\mathbf{u}$, a set of sets of possible transitions to continue $\mathbf{u}$ with
  - We require that, for finite word $\mathbf{u}$: $Acc(\mathbf{u}) \neq \emptyset$
  - Alphabet Equalization: $\Sigma \subset \Sigma'$ $\quad Acc(\mathbf{u}) \rightarrow \{ \mathbf{X} \cup (\Sigma' \setminus \Sigma) \mid \mathbf{X} \in Acc(\mathbf{u}) \}$
- $\mathbf{M} \subseteq \underline{\Sigma}^*$ where $\Sigma \subseteq \underline{\Sigma}$ is a model of $\mathbf{S} = (\Sigma, Acc)$, $\mathbf{M} \models \mathbf{S}$ iff, after equalizing alphabets:
  - For every $\mathbf{u} \in \mathbf{M} \Rightarrow \partial\mathbf{M}/\partial\mathbf{u} \in Acc(\mathbf{u})$
- Kind of a “set of automata”; implementation amounts to matching one of these automata

Residuation and Product of Acceptance Specifications

- Order is by inclusion of acceptance sets (after alphabet equalization and modulo state simulation); induces refinement
- Acceptance specification (after alphabet equalization): $\mathbf{S_1 \otimes S_2}$

$$Acc(\mathbf{S_1 \otimes S_2, u}) = \{\, \mathbf{X_1 \cap X_2} \mid \mathbf{X_1} \in Acc(\mathbf{S_1, u}) \ \wedge\ \mathbf{X_2} \in Acc(\mathbf{S_2, u}) \,\}$$

- Residuation (after alphabet equalization):
  $\mathbf{S/S_1}$ is a *pre*-acceptance spec ($Acc(\mathbf{u}) \neq, = \emptyset$)

$$Acc(\mathbf{S/S_1, u}) = \{\, \mathbf{Y} \mid \forall \mathbf{X} \in Acc(\mathbf{S_1, u})\ ,\ \mathbf{X} \cap \mathbf{Y} \in Acc(\mathbf{S, u}) \,\}$$

- Substituability of residuations

$$\mathbf{S_1 \otimes S_2 \leq S} \ \Leftrightarrow\ \mathbf{S_2 \leq clean(S/S_1)}$$

- Substituability of components

$$\mathbf{M_2 \models S/S_1} \ \text{ iff } \ \forall \mathbf{M_1}.[\ \mathbf{M_1 \models S_1} \Rightarrow \mathbf{M_1 \otimes M_2 \models S}\ ]$$

Residuation and Product of Acceptance Specifications

Observe that, even if we start with ordinary automata ($Acc(\mathbf{u})$ are all singletons), the residuation does not generally yield an ordinary automaton

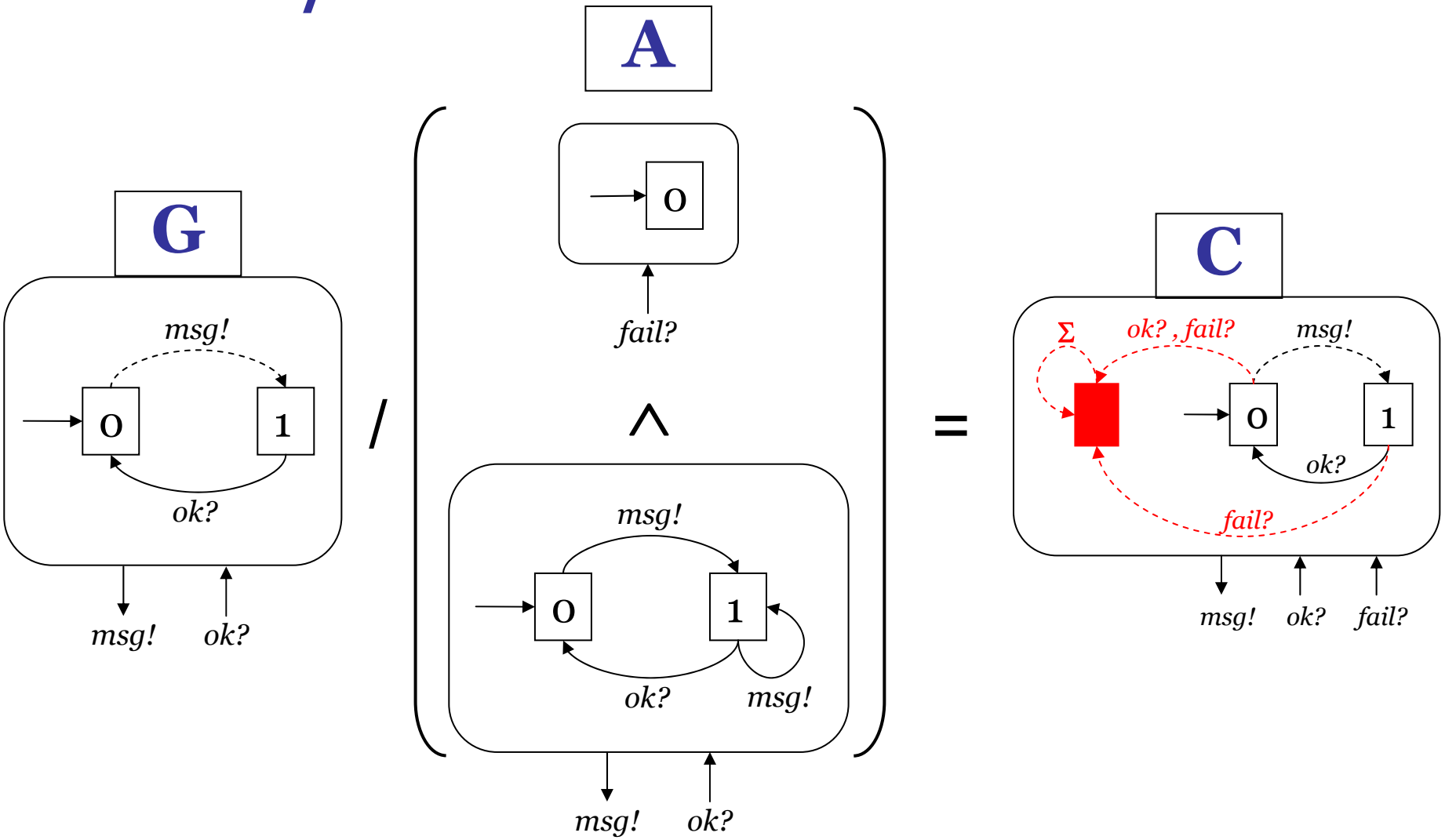
- Residuation (after alphabet equalization, see details):
  $\mathbf{S}/\mathbf{S}_1$ is a *pre*-acceptance spec ($Acc(\mathbf{u}) \neq, = \emptyset$)

$$Acc(\mathbf{S}/\mathbf{S}_1,\mathbf{u}) = \{\, \mathbf{Y} \mid \forall \mathbf{X} \in Acc(\mathbf{S}_1,\mathbf{u}) \,,\, \mathbf{X} \cap \mathbf{Y} \in Acc(\mathbf{S},\mathbf{u}) \,\}$$

$$\mathbf{S}_1 = (\Sigma_1, Acc) : Acc(\mathbf{u}) \rightarrow \quad \{\, \mathbf{X} \cup \Sigma \backslash \Sigma_1 \mid \mathbf{X} \in Acc(\mathbf{u}) \,\}$$

$$\mathbf{S} = (\Sigma, Acc) : \; Acc(\mathbf{u}) \rightarrow \quad \{\, \mathbf{X} \cup \mathbf{Y} \mid \mathbf{X} \in Acc(\mathbf{u}), \mathbf{Y} \subseteq \Sigma_1 \backslash \Sigma \,\} \cup \{\, \mathbf{Y} \mid \mathbf{Y} \subseteq \Sigma_1 \backslash \Sigma \,,\, \mathbf{Y} \neq \emptyset \,\}$$

C=G/A


G
msg!
0
1
ok?
msg!
ok?
/
A
0
fail?
∧
msg!
0
1
ok?
msg!
msg!
ok?
=
C
Σ
ok? , fail?
msg!
0
1
ok?
fail?
msg!
ok?
fail?

Contracts as pairs $\{A,C\}$ using residuation

- Observe that, in the above example $C=G/A$, we have that G is compatible with A, that is, for every pair (p_G,p_A) of states, we have

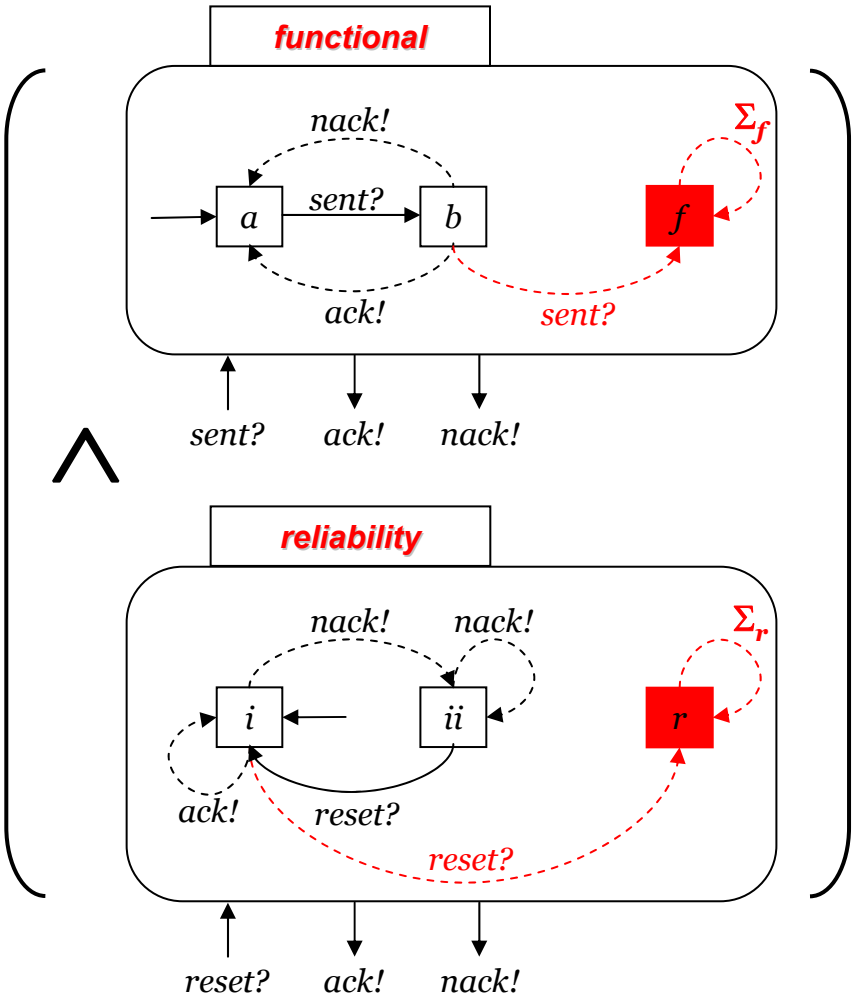
$$must_G(p_G) \cap mustnot_A(p_A) = \emptyset$$

- If the designer specifies a pair $\{A,G\}$, such a property may get violated, making G and A incompatible
- Thus, in general, the right formula for the resulting contract is

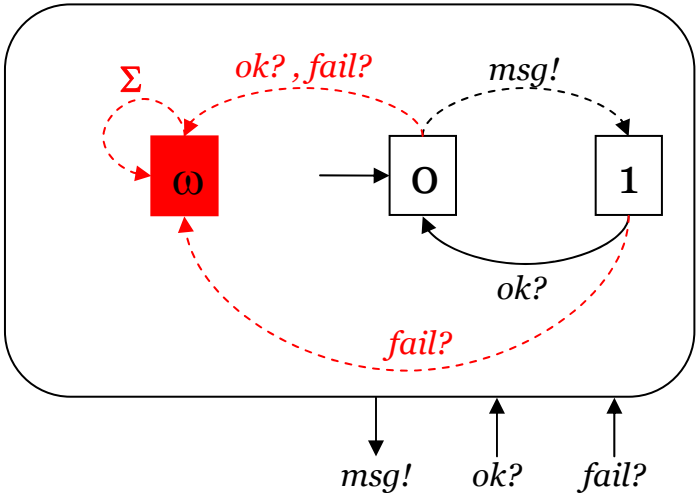
$$C = (G \otimes A)/A$$

Now that we have residuation at hand, can we use it to perform synthesis?

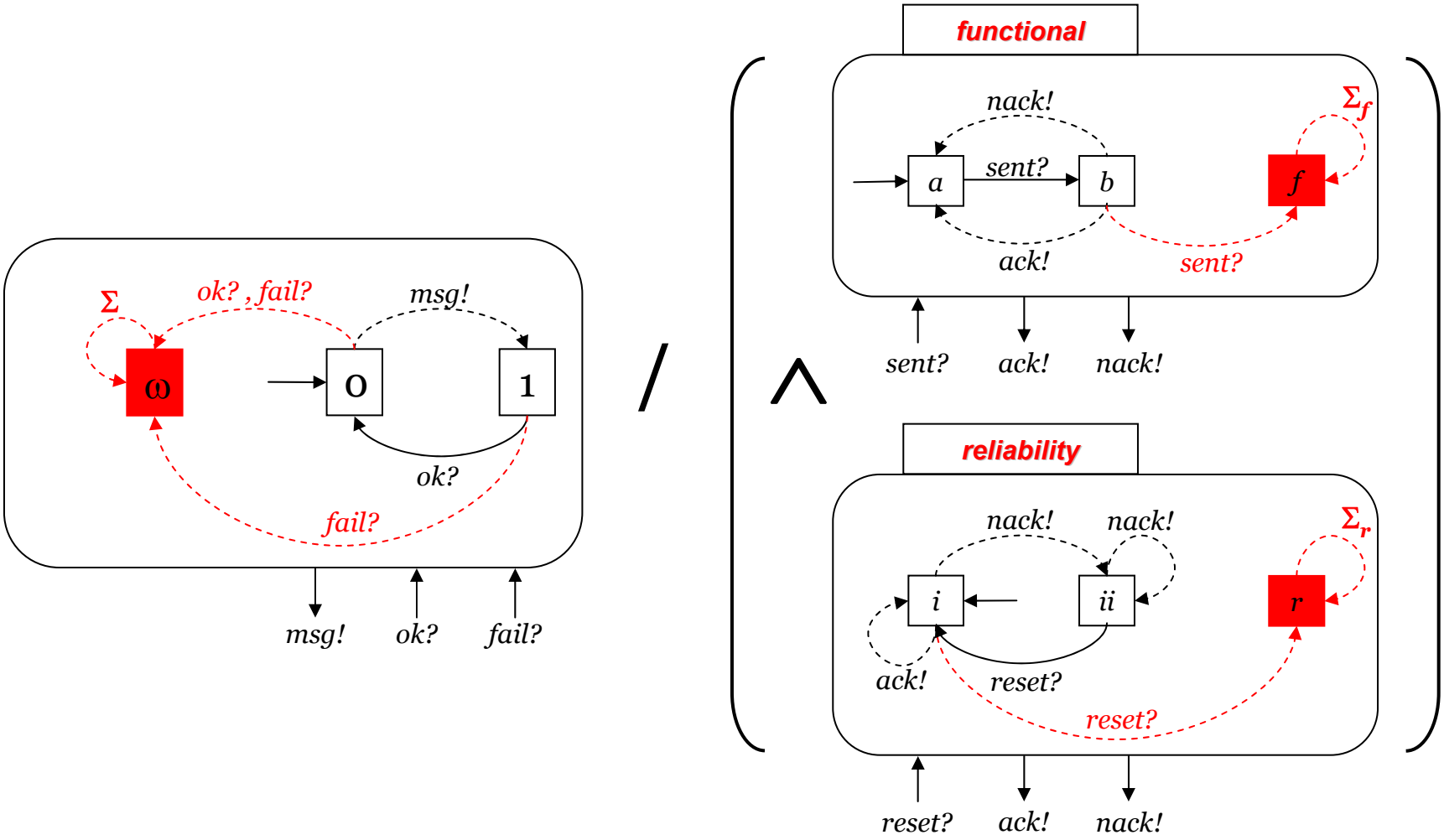
Protocol spec as residuation *client/netw*?



$\otimes ? =$



Protocol spec as residuation *client/netw*?



(z|w)/(a,i|a,ii|b,i|b,ii|f,i|f,ii|a,r|b,r|f,r), top :

{ X | X ⊆ {msg, ok, fail, ack, nack, reset, sent}}

u/(a,i|a,r) :

{{ok} ∪ X, {sent} ∪ Y | X ⊆ {fail, ack, nack, reset, sent}
Y ⊆ {ack, nack, reset}}

u/(a,ii) :

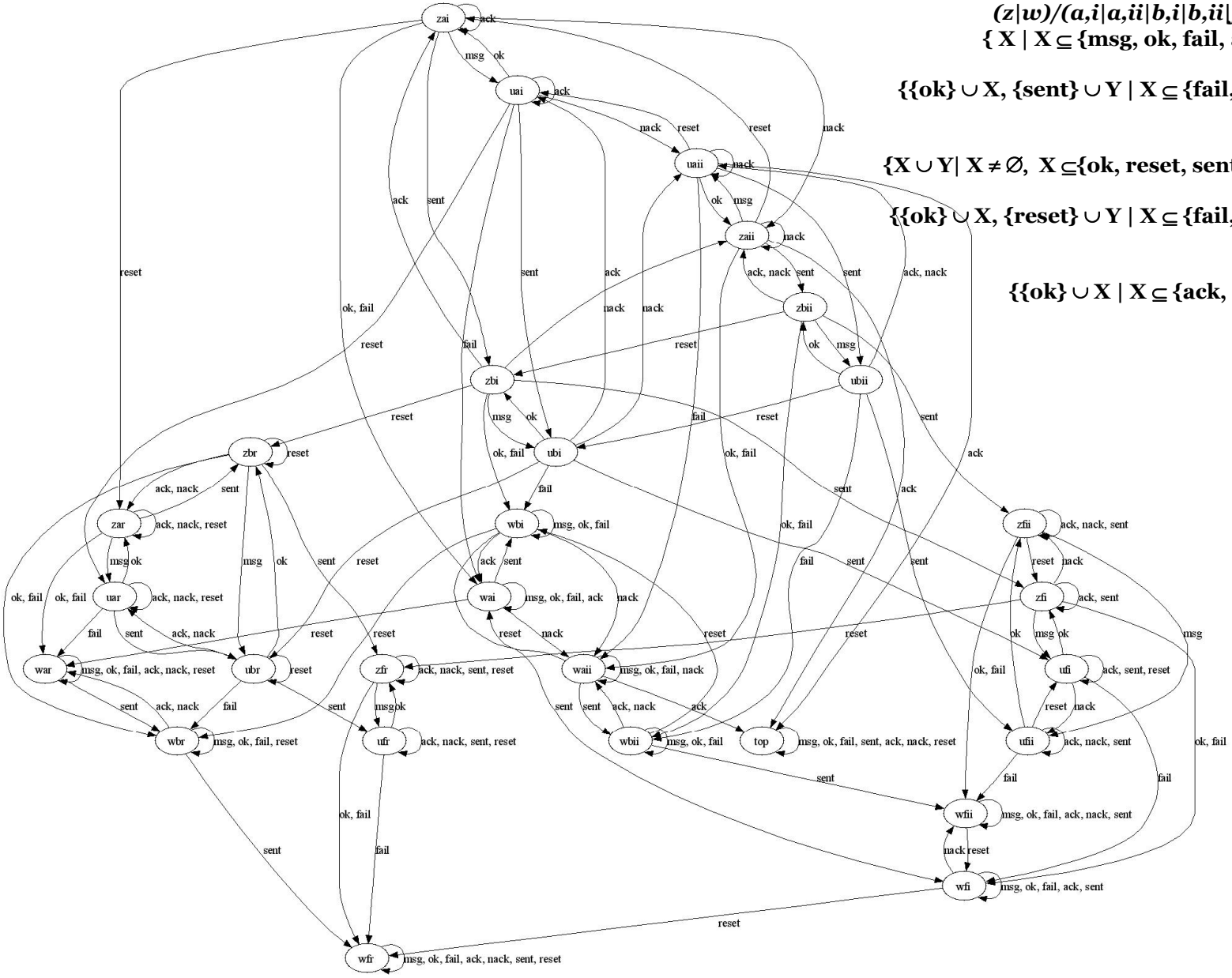
{X ∪ Y| X ≠ ∅, X ⊆{ok, reset, sent}, Y ⊆ {ack,nack, fail}}

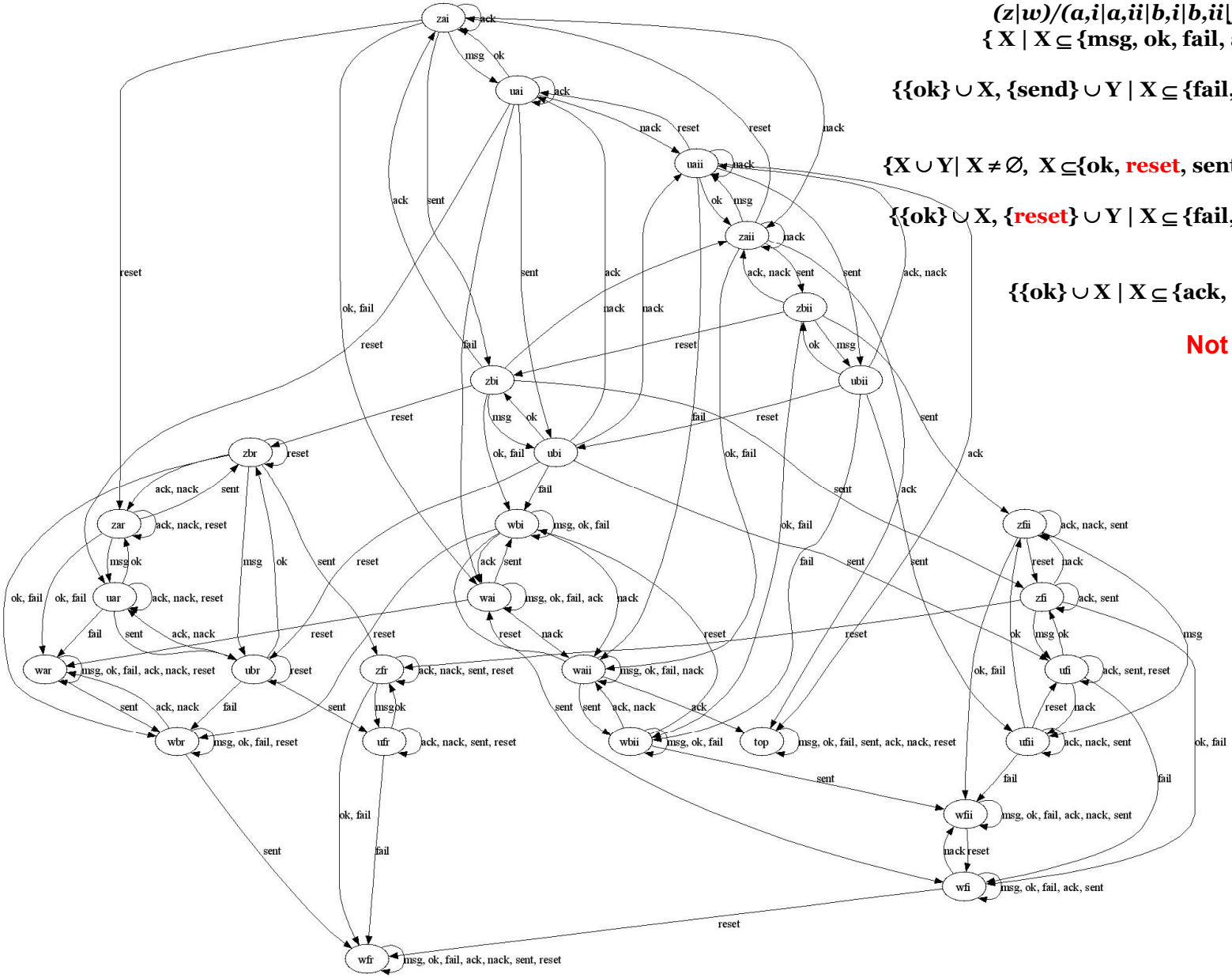
u/(b,ii|f,ii) :

{{ok} ∪ X, {reset} ∪ Y | X ⊆ {fail, ack, nack, reset, sent}
Y ⊆ {ack, nack, sent}}

u/(b,i|f,i|b,r|f,r) :

{{ok} ∪ X | X ⊆ {ack, nack, fail, reset, sent}}





(z|w)/(a,i|a,ii|b,i|b,ii|f,i|f,ii|a,r|b,r|f,r), top :

{ X | X ⊆ {msg, ok, fail, ack, nack, reset, sent}}

u/(a,i|a,r) :

{{ok} ∪ X, {send} ∪ Y | X ⊆ {fail, ack, nack, reset, sent}
Y ⊆ {ack, nack, reset}}

u/(a,ii) :

{X ∪ Y| X ≠ ∅, X ⊆{ok, reset, sent}, Y ⊆ {ack,nack, fail}}

u/(b,ii|f,ii) :

{{ok} ∪ X, {reset} ∪ Y | X ⊆ {fail, ack, nack, reset, sent}
Y ⊆ {ack, nack, sent}}

u/(b,i|f,i|b,r|f,r) :

{{ok} ∪ X | X ⊆ {ack, nack, fail, reset, sent}}

Not part of the Protocol!

The resulting specification

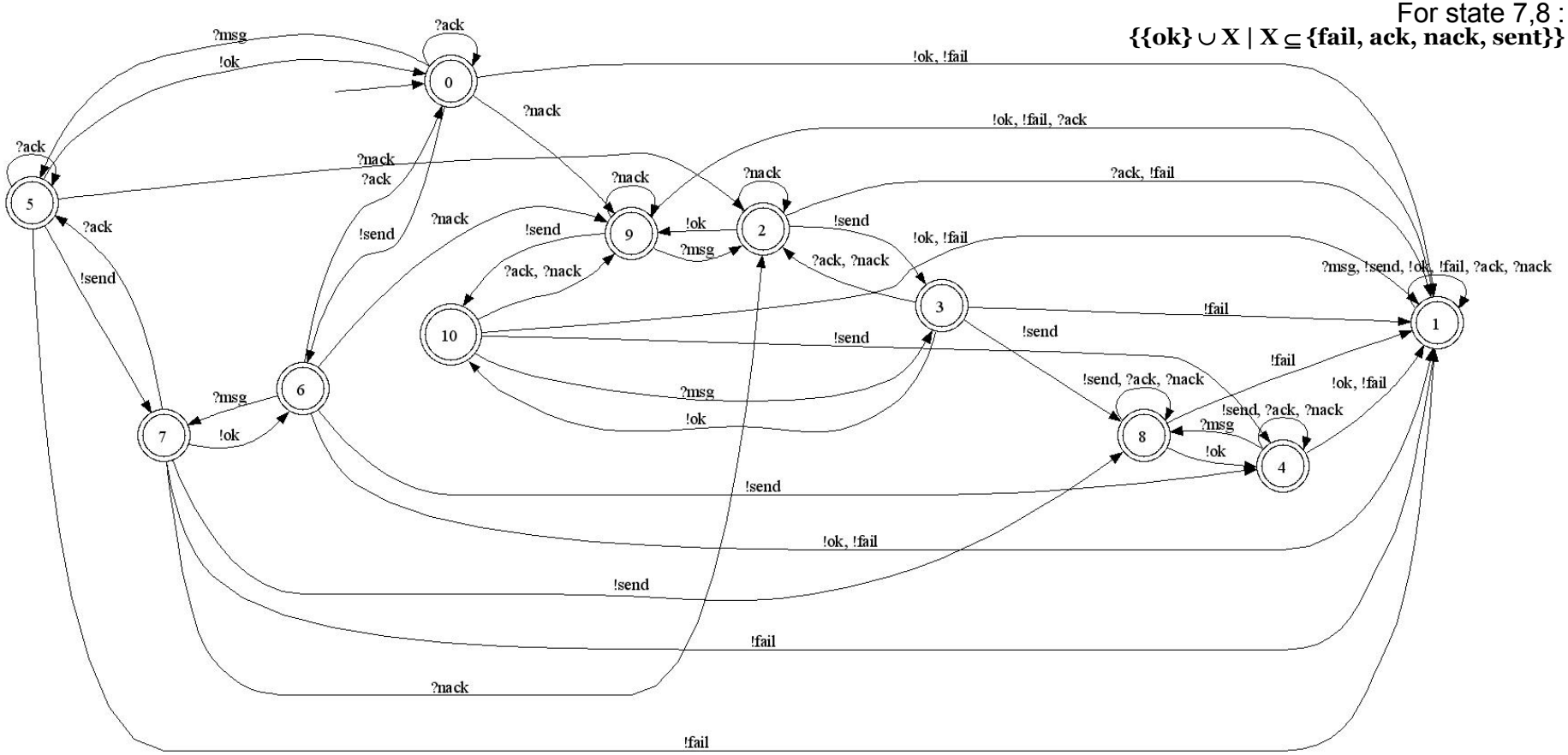
For states 0,1,4,6,9,10 :
$\{ X \mid X \subseteq \{msg, ok, fail, ack, nack, sent\}\}$

For state 2 :
$\{X \cup Y \mid X \neq \emptyset,\ X \subseteq \{ok, sent\}, Y \subseteq \{ack, nack, fail\}\}$

For state 3 :
$\{\{ok\} \cup X, Y \mid X \subseteq \{fail, ack, nack, sent\}, Y \subseteq \{ack, nack, sent\}\}$

For state 5 :
$\{\{ok\} \cup X, \{send\} \cup Y \mid X \subseteq \{fail, ack, nack, sent\}, Y \subseteq \{ack, nack\}\}$

For state 7,8 :
$\{\{ok\} \cup X \mid X \subseteq \{fail, ack, nack, sent\}\}$



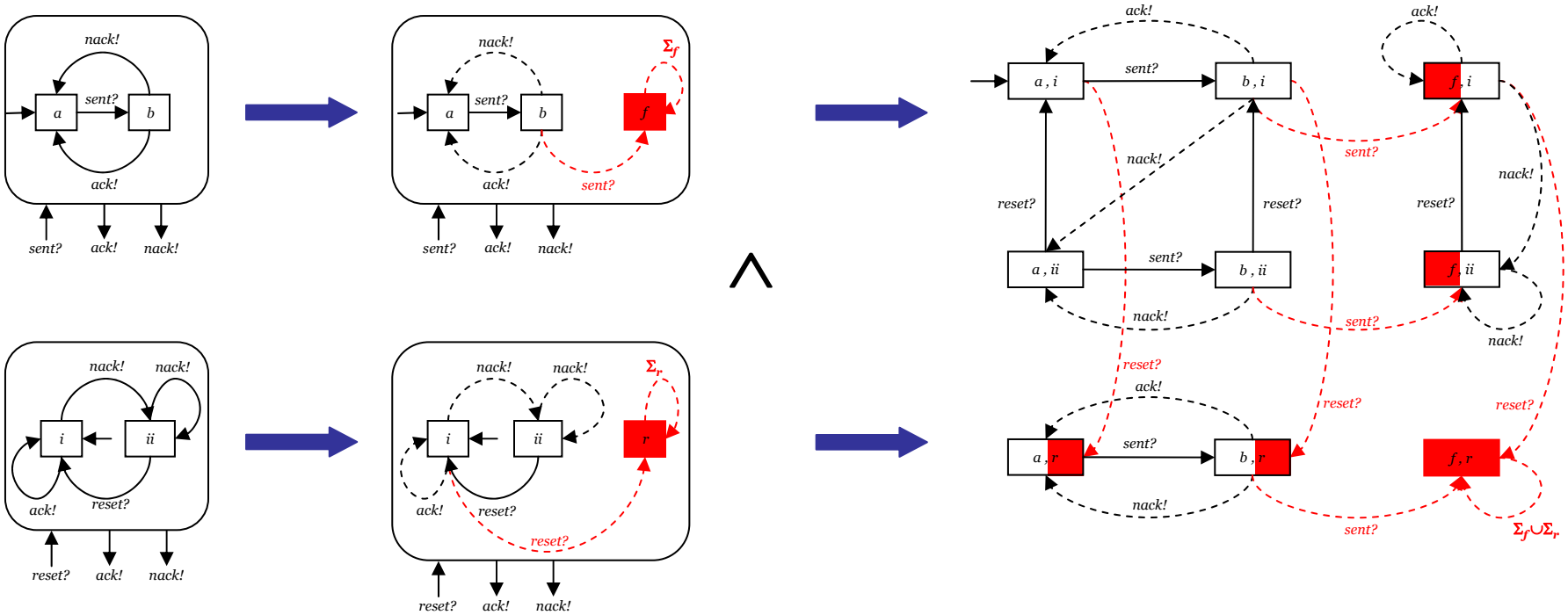
Summary

A systematic approach

Interface Automata

⇓

Contracts = Modal Automata with traps & Combinators

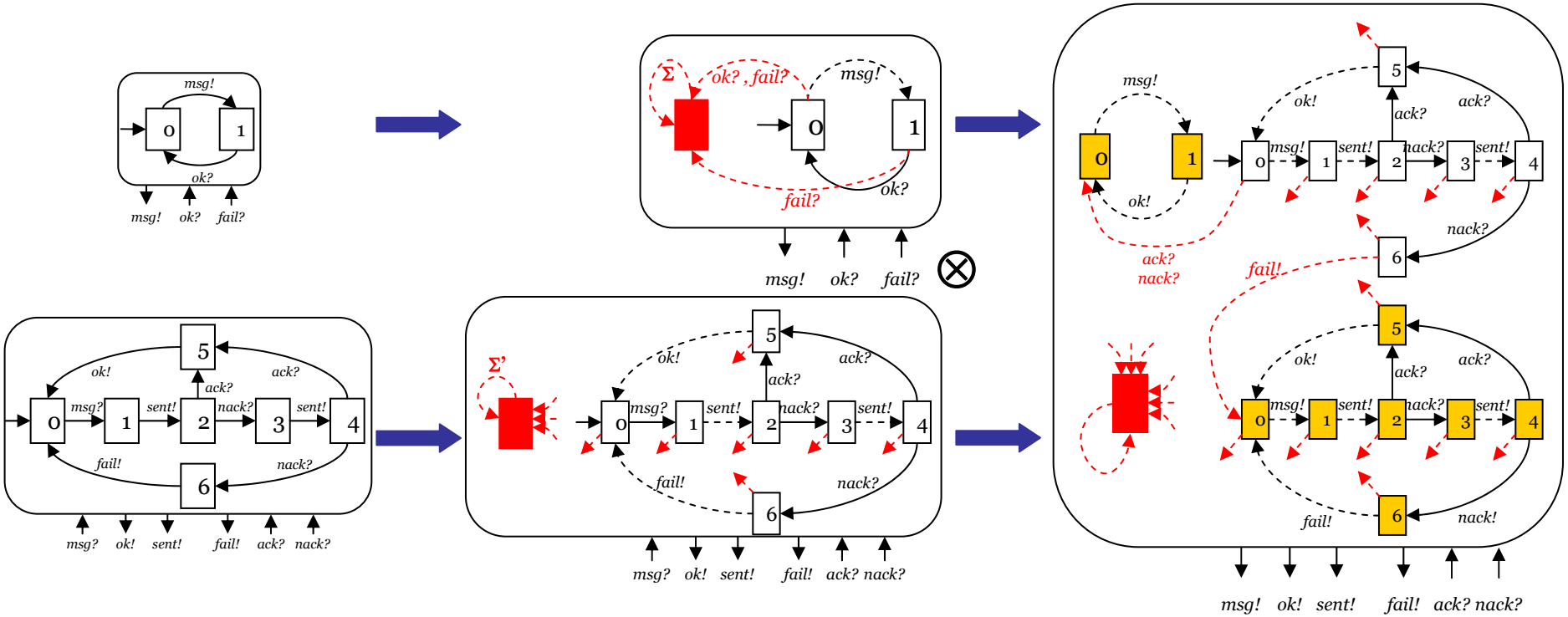


A systematic approach

Interface Automata

⇓

Contracts = Modal Automata with traps & Combinators



A systematic approach

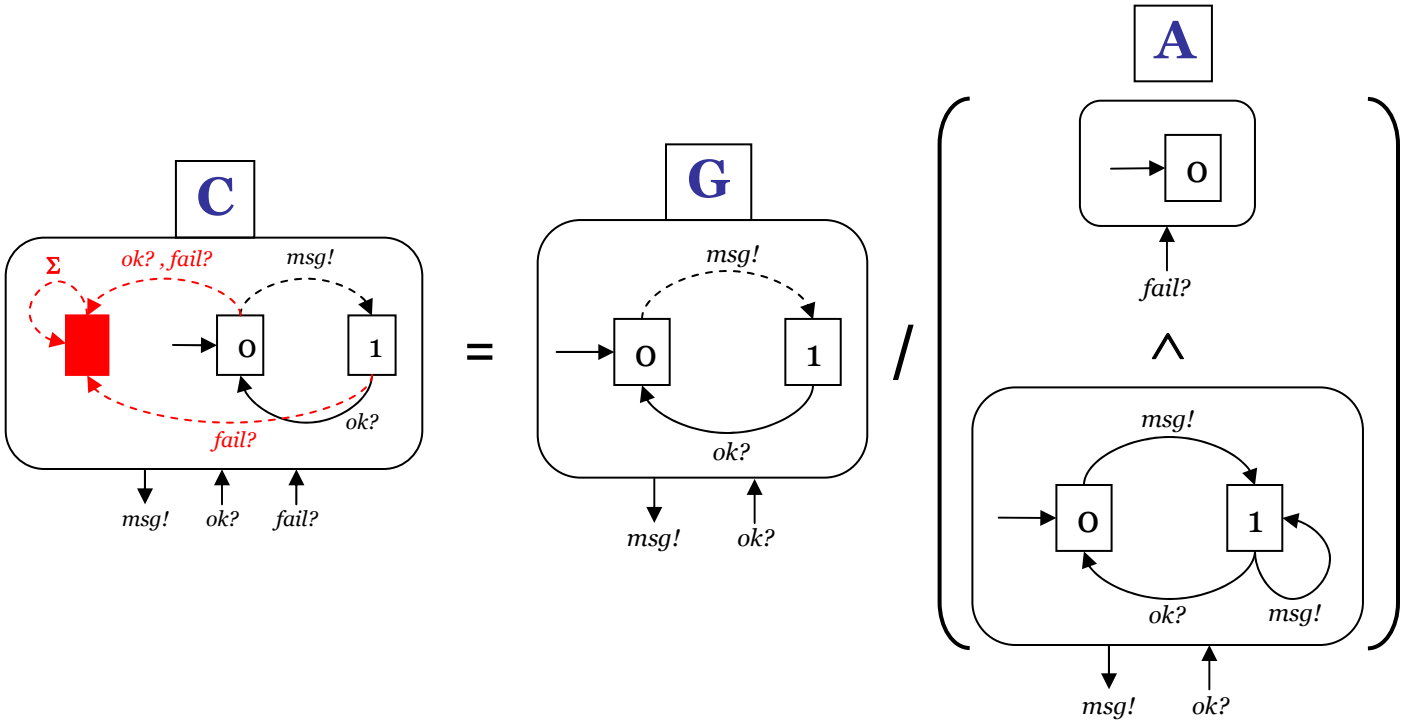
Interface Automata

⇓

Contracts = Modal Automata with traps & Combinators

⇓

G/A



A systematic approach

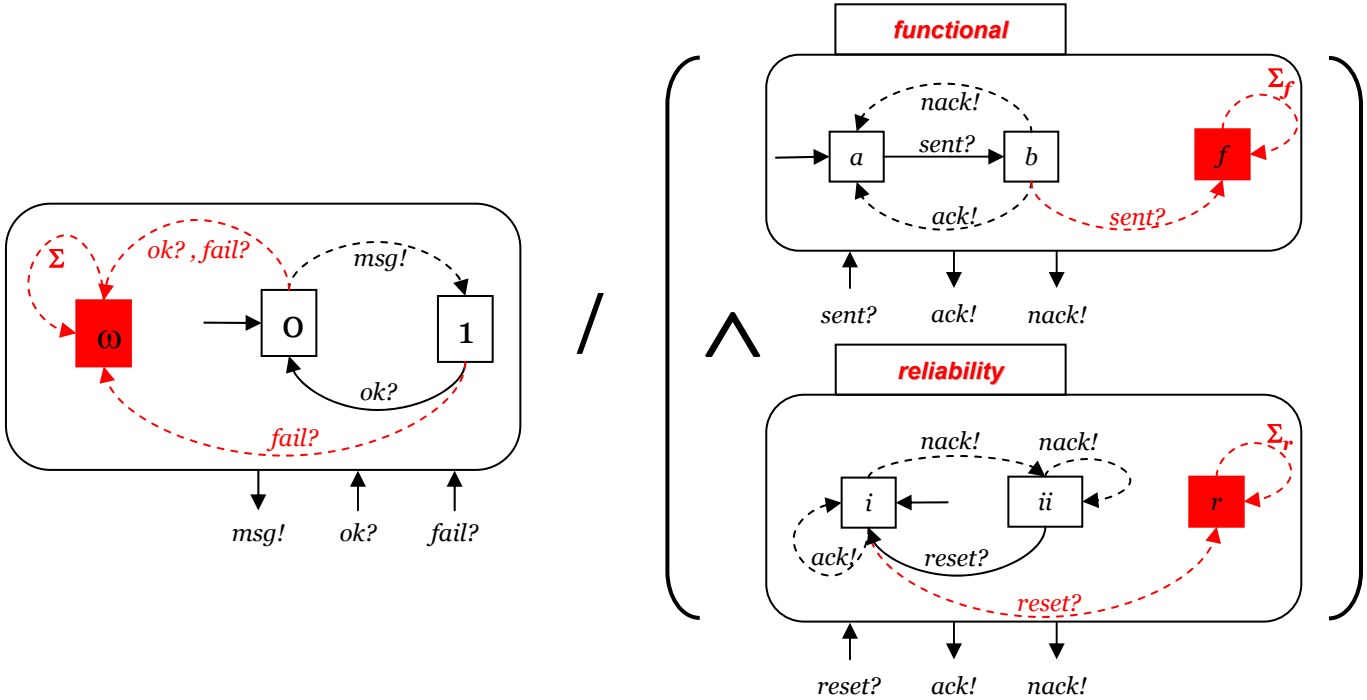
Interface Automata

⇓

Contracts = Modal Automata with traps & Combinators

⇓

Contract Synthesis



Concluding Remarks, Frustrations, Perspectives

40 ans la révolution de l'information

INRIA

Concluding remarks

- In component based design of complex embedded systems, multiple viewpoints/aspects must be considered
- This required revisiting concepts of component, contract, and interface.
- We have done this by enhancing de Alfaro – Henzinger Interface Automata, called AG-automata: just explicit what happens when Assumptions are violated by the environment.
- Formalization relied on Modal Automata (Kim Larsen) and Acceptance Automata (J-B Raclet). The latter provided the adequate algebraic support by offering the needed embedding of automata to support contract composition, residuation, and implementation, ensuring substituability and reusability of components.
- Flexibility in design flow was studied (components first or viewpoints first).

Frustrations

- The study does not seem complete: status of partially-red states yet unclear
- What is best from user's viewpoint:
  - a "single-diagram" contract specification blending A's and G's?
  - an explicit decomposition as C=G/A?
    but it may be that G's and A's have to speak about everything…
    at least: is A "smaller" than C?

Perspectives

- So far we considered AG-automata (or modal automata with traps)
- Not appropriate to address heterogeneity: we need richer frameworks instead, cf. the machine model proposed in SPEEDS project (quite straightforward in fact):
  - supports dataflow style of modeling
  - allow interactions simultaneously involving multiple ports
  - symbolic transition systems with variables
  - complex transitions, possibly involving continuous dynamics, probabilities…
- Well, we believe that developing the needed theories should be doable