Parameterised Contracts

for Software-Component Protocols

Ralf Reussner

reussner@ira.uka.de
LIIN
Universität Karlsruhe (TH)

Dec, 21st. 2000

Overview

1. Scenarios of Software Component Development
2. Example: Extensible Mail System
3. Trusted Components
4. Parameterised Contracts
5. Interfaces, Contracts, and Types
6. Types for Components
7. Type Adaption
8. Type Extension
9. Meta-Protocol
10. Adaptor generation: Join, Split, and Protocol Changers
11. Hiding FSMs
12. Conclusions
13. Future Work

1. Scenarios

Component Developer Scenario:

Functionality vs. Reusability: Large functionality:

- (a) interesting for many users,
- (b) high dependencies to environment.

$\rightarrow$ Functionality/Reuse-Dilemma

Extension of Functionality: Hard to anticipate complete functionality of a component during design.

1. Scenarios

System Architect Scenario:

Early Evaluation of architectures in terms of:

- Achieved degree of reuse / Effort for adaptation
- Non-functional properties: reliability, timing, security

Adaptation: Reuse of blackbox-components

Early Error Detection: Missing contextual information hinders error detection during compile-time.

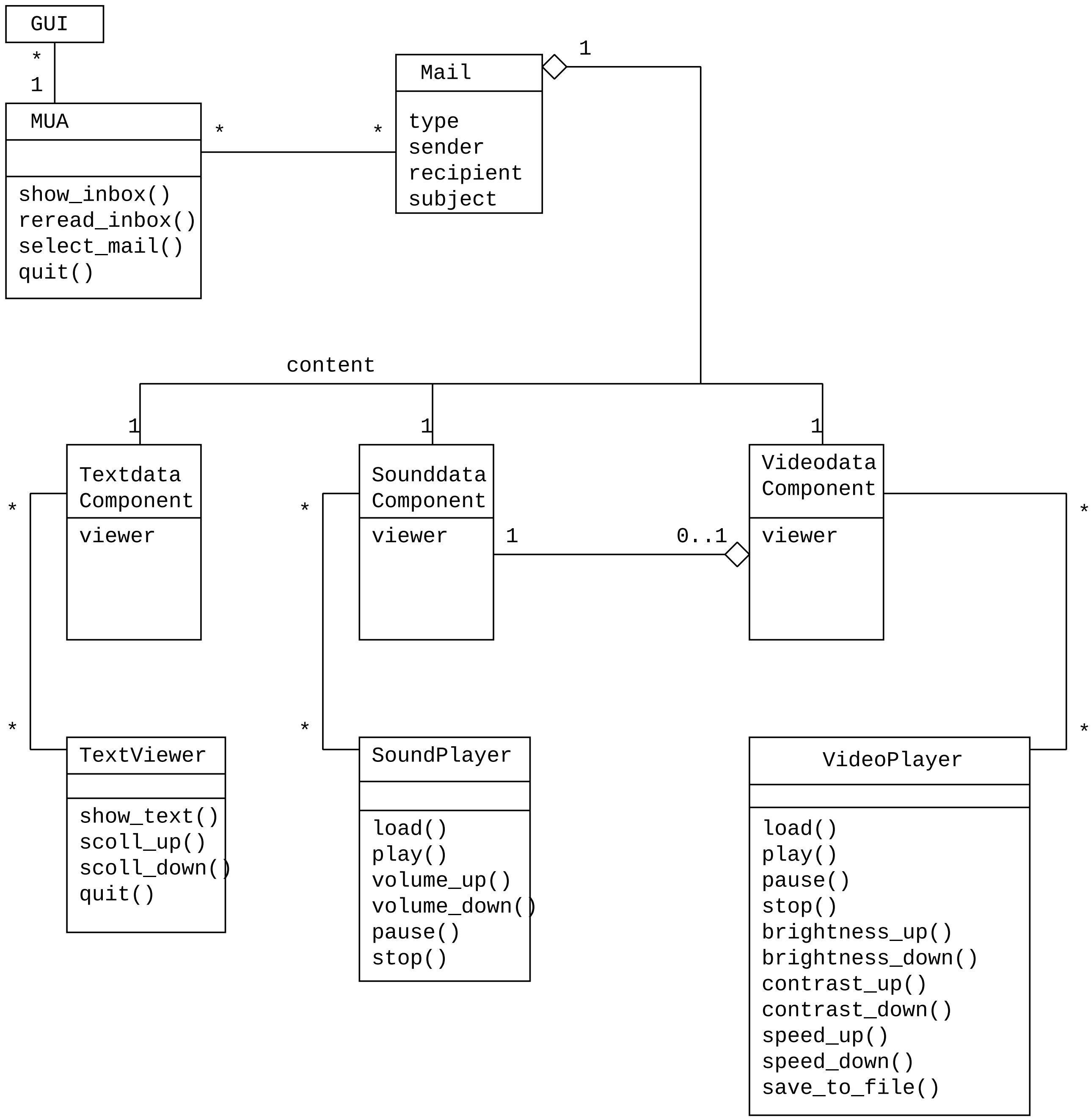
Incompatibilities: Incompatible signatures and protocols often hinder interoperability.

System Administrator / User Scenario:
Expects preservation of system properties during reconfiguration:

- Functionality
- Absence of compositional errors
- Non-functional properties: reliability, timing, security

2. Example System

MailUserAgent with different kinds of mails
and multiple players/viewers:



3. Trusted Components

Trusted Components:

Deduction of System Properties out of components properties

- Exclusion of compositional errors
- Static reasoning about components properties

Secure Extension: Preservation of component properties during extension (with plug-ins).

- **Security**
- Real-time properties

3. Trusted Components

Trusted Components:

Adaptation: parameterised contracts

- Includes adaptor generation if necessary / possible.
- Evaluation of architectures through hierarchical component model.

Review: Normal Contracts

- Precondition: fulfilled by client (caller)

- Postcondition: fulfilled by supplier (callee)

$$\underbrace{\texttt{float}}_{\text{Post-condition}}\ \texttt{sqrt}\ (\underbrace{\texttt{float}}_{\text{Pre-condition}}\ \texttt{a})$$

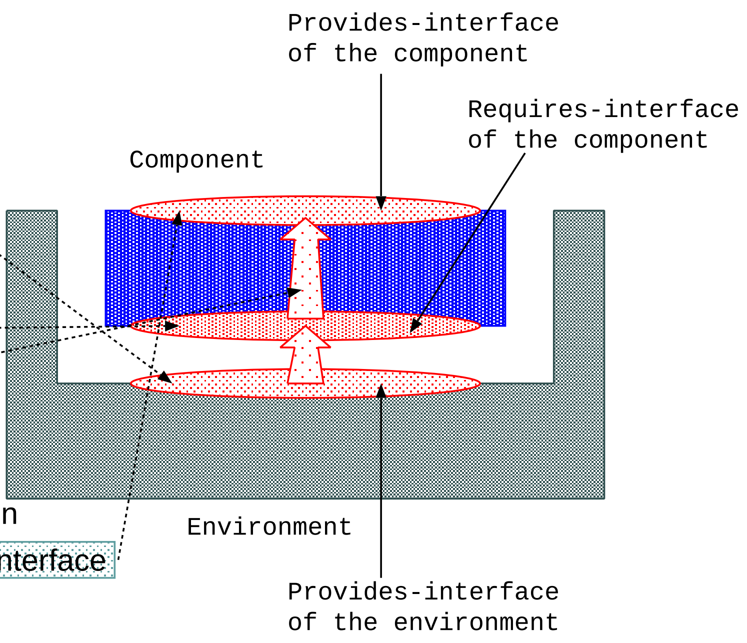
Normal Contract:
if precondition **then** postcondition

4. Parameterised Contracts

Normal Contract: Postcondition is a boolean function of precondition.

Parameterised Contract: Postcondition is a computed in dependence of the precondition during composition time.

When a component is inserted in an environment and recognizes that the provides-interface of this environment does not offer all required functionality (as described in the requires-interface of the component) than the connection of the provides- and requires-interface of the component allows the adaption of the component's provides-interface in such a way, that the requires-interface of the component fits to the provides-interface of the environment.



Parameterised Contracts: Mapping preconditions to postconditions.

5. Interfaces, Contracts, and Types

All the same?

Documentation: All information about a component.

Interface: Machine processable documentation concerning the component's application without inspecting the component's code.

Contract: Interface information specifying the required and provided computation.

Type: Special contract, simplified for the sake of static checkability.

6. Types for Components

Our Goal: A Type System for Software-Components
to support:

Adaption: Elimination of environmental dependencies through adaption of functionality according to the capabilities of the environment.

Extension: Defined Interfaces to plug-ins.

Error Detection during composition-time: Detection of composition errors before run-time.

Adaptor Generation: (Semi-)automatic generation of adaptors out of interface information.

In this talk:

Type System regarding Service Availability (Component Protocols).

Protocol information of a component

Availability of services: (*provided-interface*)

- Given in a protocol: To use service `read`, first call service `open`, ...
- Given by permissions: Restriction of services due to security issues.
- Given by the resources of the infrastructure: A component can only offer services, which are supported by the underlying infrastructure (other components, operating system, hardware).

Dependencies to infrastructure: (*required-interface*)

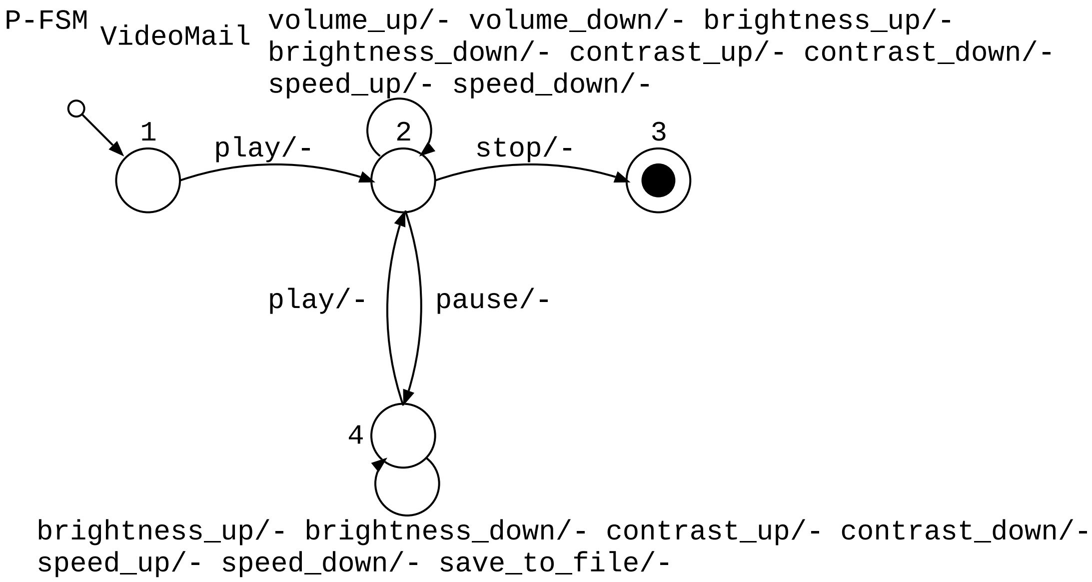
- Which other components are required?
- How is a required component used? (correct usage of the the required component's provides interface)

Protocol issues are modeled by *enhanced FSMs*.

6. Types for Components

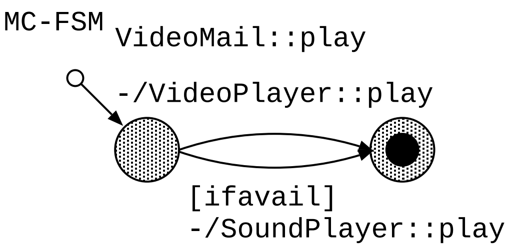
FSMs to model the Type of a Component

Provides-protocol: P-FSM$_{\text{VideoMail}}$



Requires-protocol: MC-FSMs

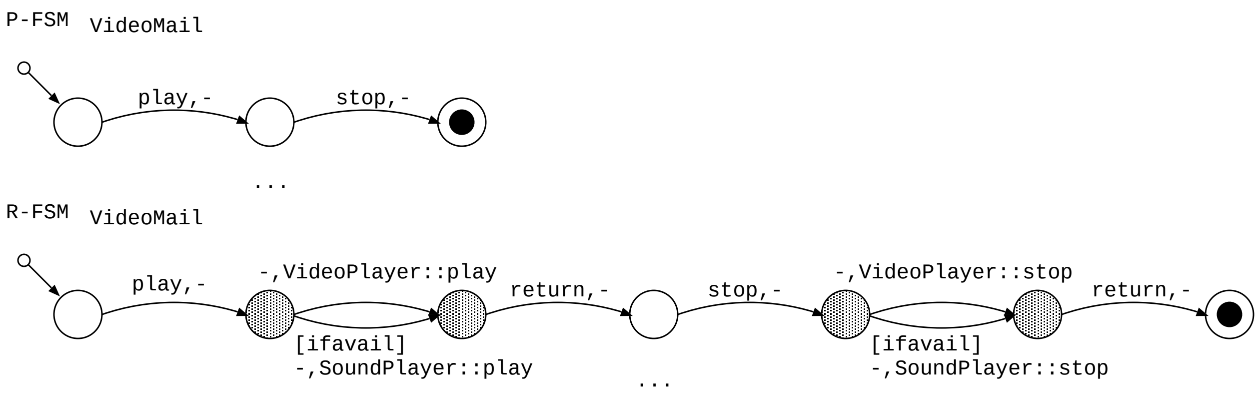
Description of the usage of external functions for each *function*



6. Types for Components

Requires-protocol:

R-FSM$_{\text{VideoMail}}$



- Insertion of the MC-FSMs into the P-FSM.
- Explicit marking of calls and returns of functions.
- Mapping of provided functionality to required functionality.

6. Types for Components

Why FSMs?

- Elaborated theoretical results about FSMs

Normalised form:	unique minimal FSM
	Test of Isomorphism *in this case* simple.
Subtype property:	Subset relation of regular languages
	Comp B is Subtype of Comp A $\Leftrightarrow$
	$L(\text{P-FSM}_B) \supseteq L(\text{P-FSM}_A) \wedge$
	$L(\text{R-FSM}_B) \subseteq L(\text{R-FSM}_A)$

- Algorithms for FSMs combinable with graph algorithms

7. Type Adaption

Motivation:

- Large Functionality of a component often translates into strong dependencies (requirements) of this component to its environment.
- Adaption allows a subset of a component's functionality even under insufficient environmental resources.
- VideoMail ($\text{P-FSM}_{\text{VideoMail}}$, M-FSMs) adapts accordingly when coupling with VideoPlayer ($\text{P-FSM}_{\text{VideoPlayer}}$)

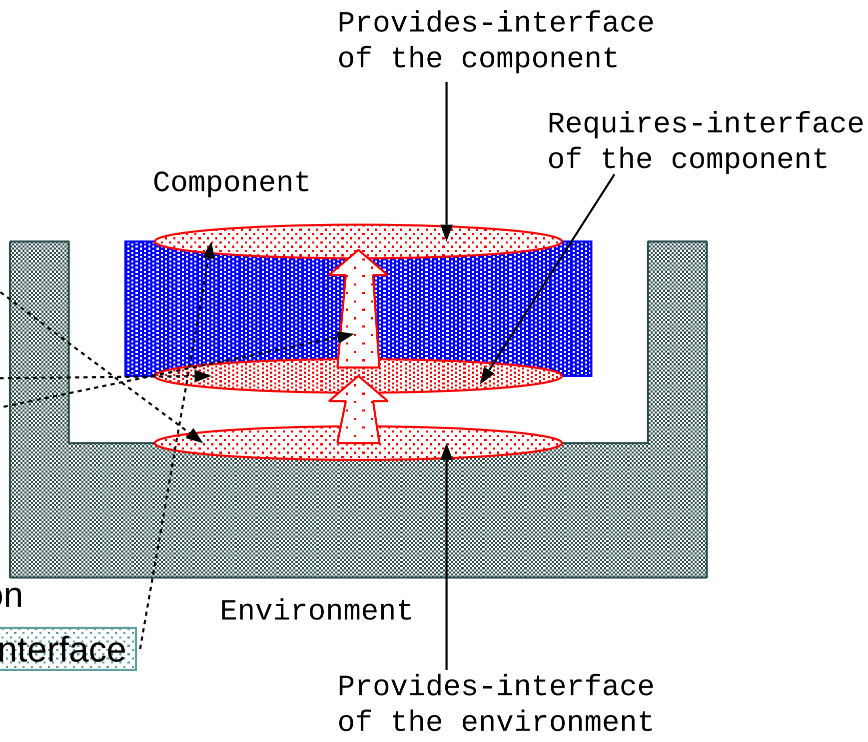
Algorithm:

- Compute $\text{R-FSM}_{\text{VideoMail}}$
- Compute crossproduct
  $\text{R-FSM}_{\text{VideoMail}\times\text{VideoPlayer}} :=$
  $\text{R-FSM}_{\text{VideoMail}} \times \text{P-FSM}_{\text{VideoPlayer}}$
- Compute $\text{P-FSM}_{\text{VideoMail}\times\text{VideoPlayer}}$ out of $\text{R-FSM}_{\text{VideoMail}\times\text{VideoPlayer}}$

7. Type Adaption

Schema:

When a component is inserted in an environment and recognizes that the provides-interface of this environment does not offer all required functionality (as described in the requires-interface of the component) than the connection of the provides- and requires-interface of the component allows the adaption of the component's provides-interface in such a way, that the requires-interface of the component fits to the provides-interface of the environment.

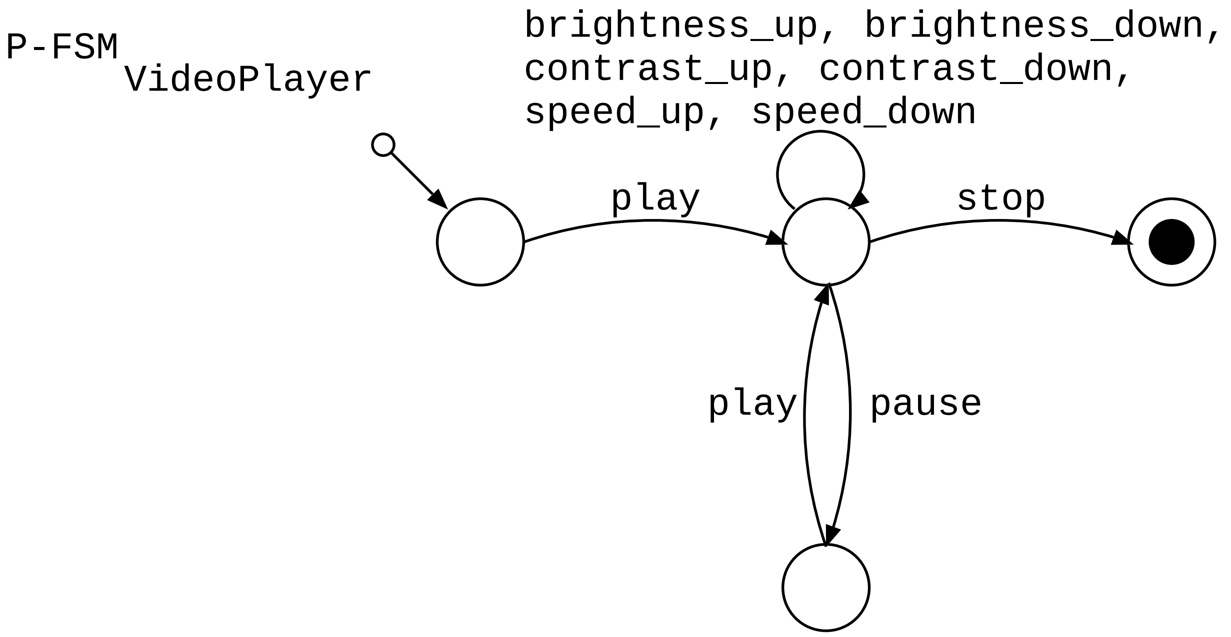


Parameterised Contracts: Mapping preconditions to postconditions.

7. Type Adaption

Example

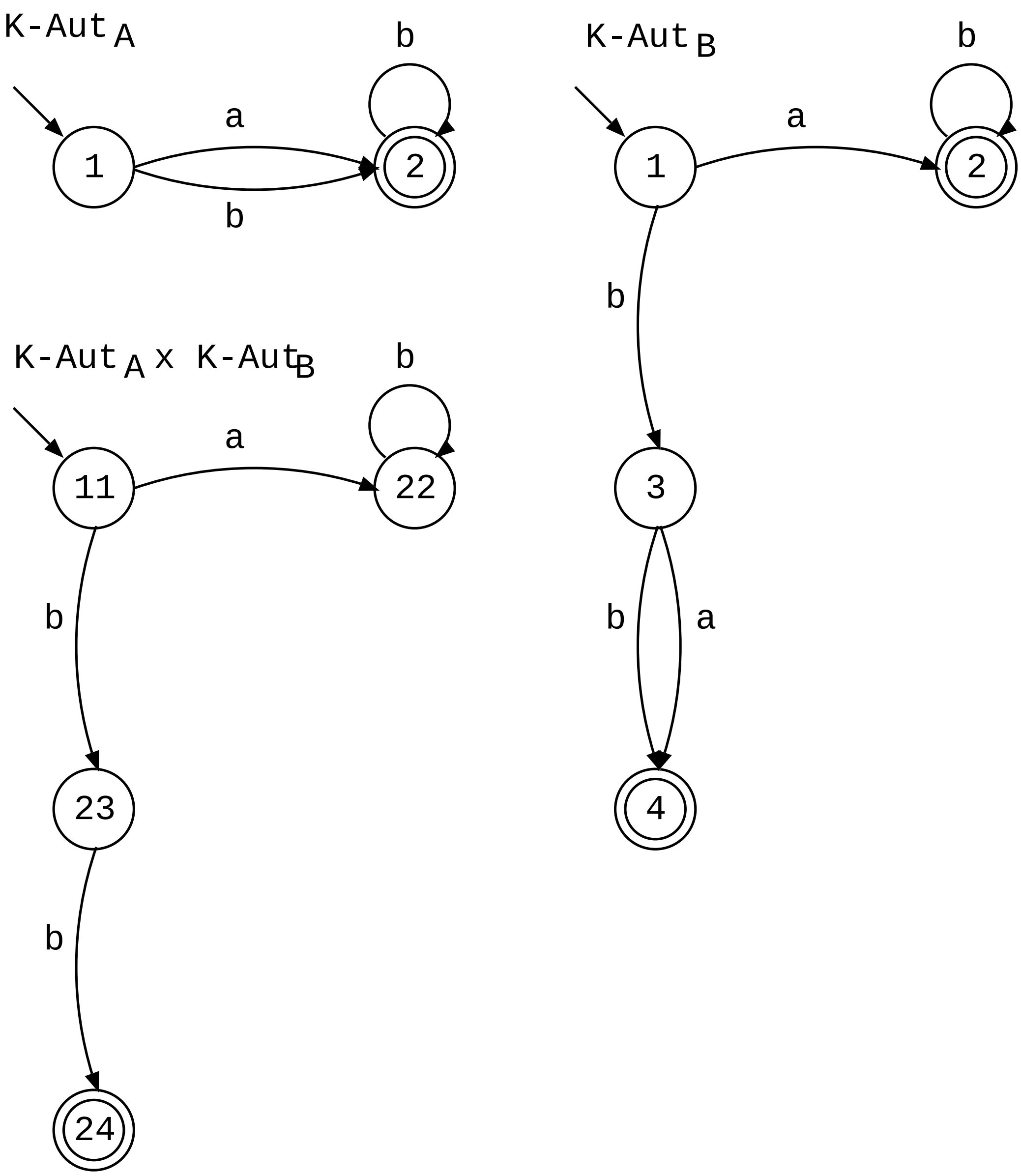
The VideoPlayer to couple with:



7. Type Adaption

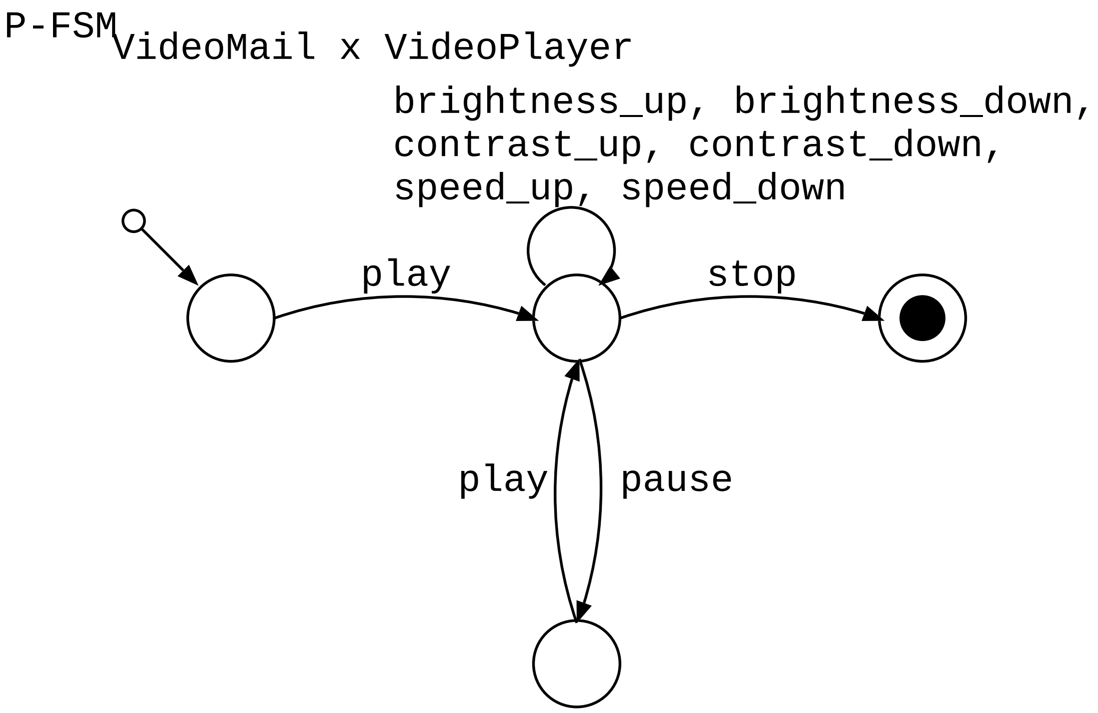
Example

How to build crossproducts:



7. Type Adaption

Example



8. Type Extension

Motivation:

- MailUserAgent cannot know all types of mails in advance.
  $\Rightarrow$ it changes its type (P-FSM$_{\text{MailUserAgent}}$, M-FSMs) when extended with the VideoMail (P-FSM$_{\text{VideoMail} \times \text{VideoPlayer}}$)

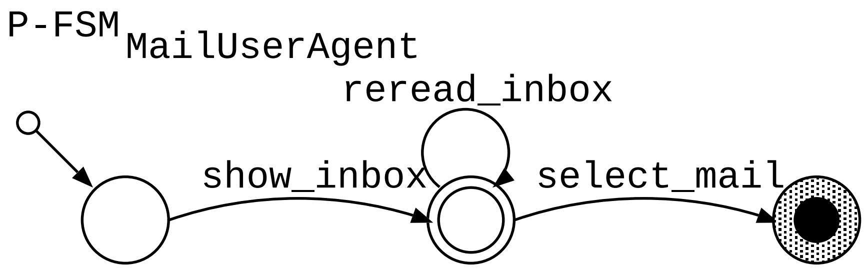
- What is the type afterwards?
  (P-FSM$_{\text{MailUseragent} + (\text{VideoMail} \times \text{VideoPlayer})}$)

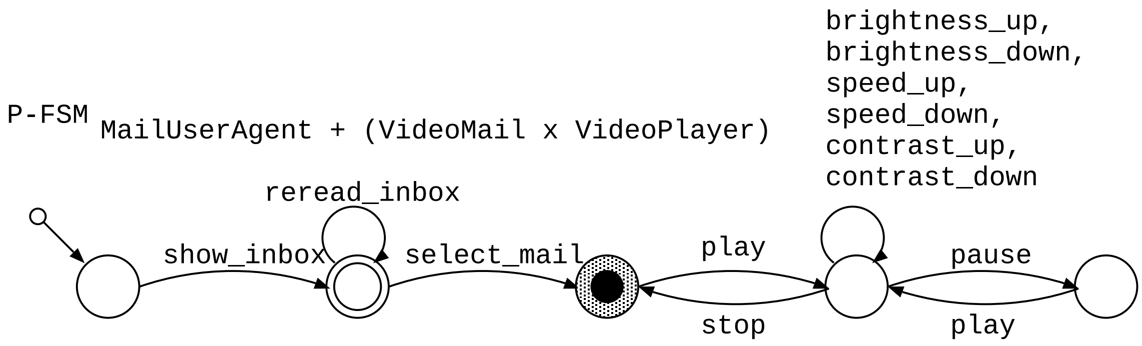
Solution:

Insertion of P-FSM$_{\text{VideoMail} \times \text{VideoPlayer}}$ into P-FSM$_{\text{MailUseragent}}$ at defined *coupling points*.

8. Type Extension

Example





8. Type Extension

Statically Checkable Constraints:

- Required minimal functionality (like constraint parametric polymorphism)
  e.g., at least method `play` is required.
- Exclusion of unwanted functions (like sandbox model)
  e.g., function`format` must be excluded at all.
- Exclusion of unwanted functionality
  e.g., sequences like, ’`fileRead`’ or ’`fileWrite`’ or ’`unlink`’ must be excluded
  But: sequences like ’`userDialog - fileRead`’ are allowed

Solution: Plug-In-specific definition of:

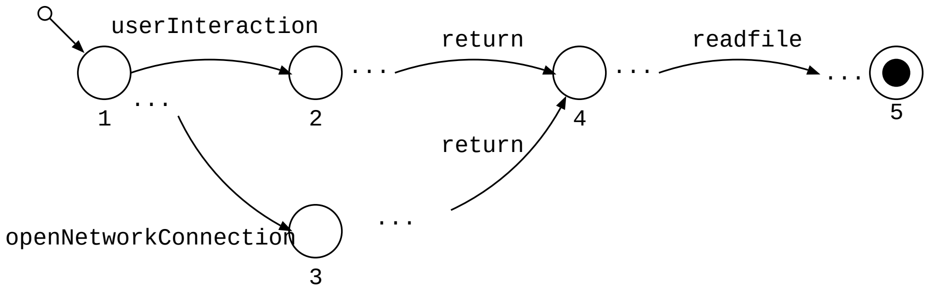
E Set of Excluded sequences

R Set of Required sequences

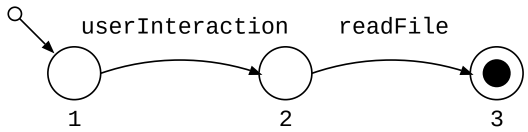
A Set of Allowed sequences

Example for Exclusion: Attachment-Sender

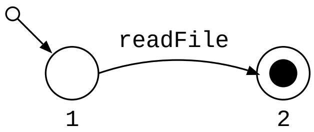
R-FSM
AttachmentSender



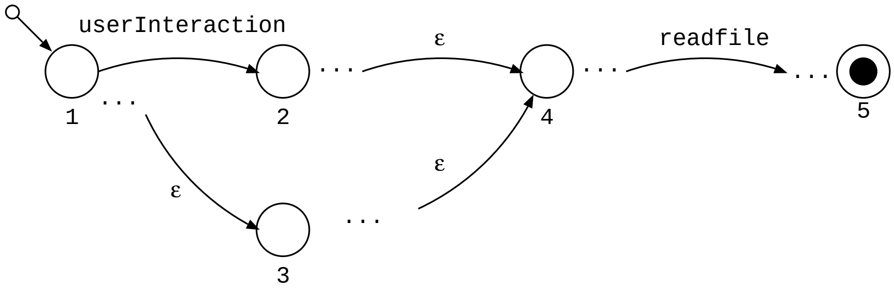
A-FSM
AttachmentSender



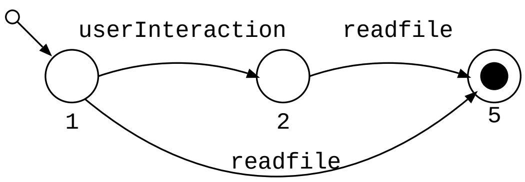
E-FSM
AttachmentSender



epsilon-FSM
AttachmentSender

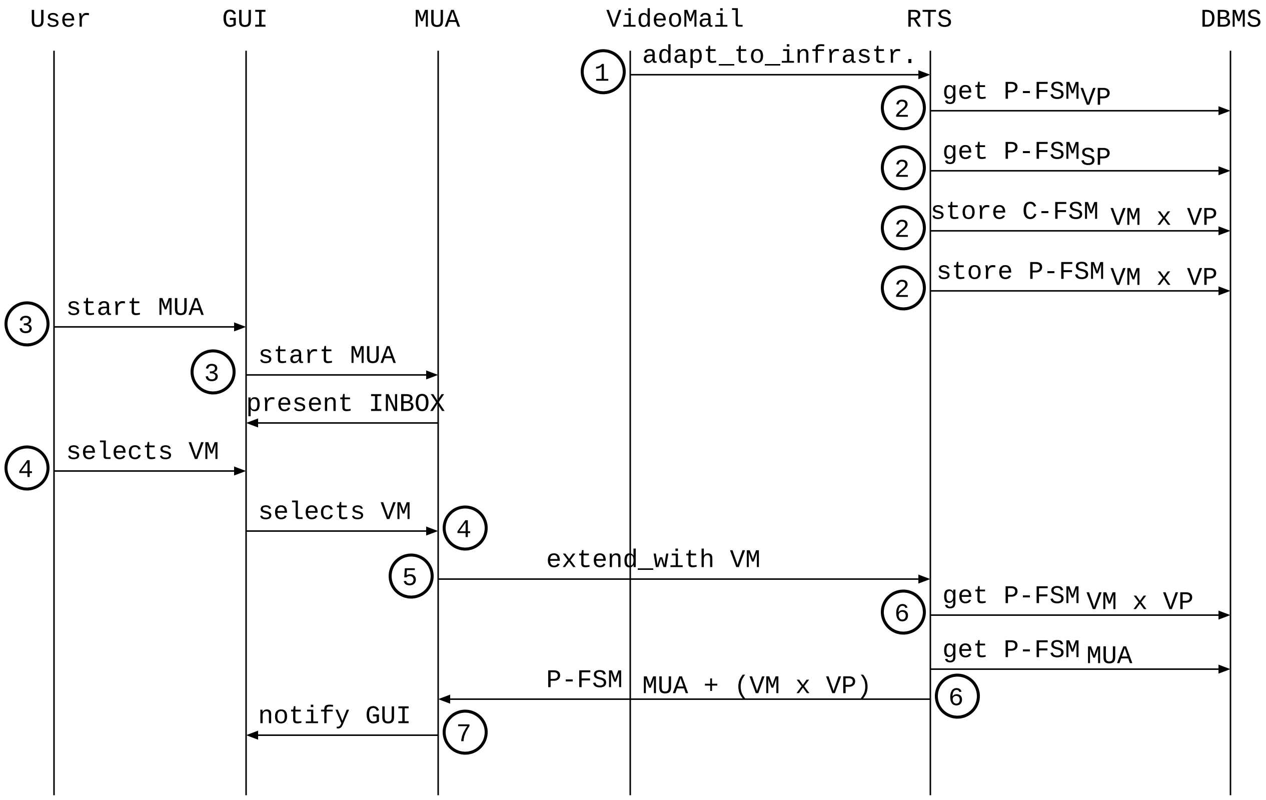


Minimiert:



1. ε-setzen aller Methoden aus $I_{R\text{-}FSM} \setminus I_{A\text{-}FSM}$
2. Minimieren
3. Test (M \ A-FSM) $\cap$ E-FSM == 0 ?

9. Meta-Protocol



Legend:

DBMS	DataBase-ManagementSystem	MUA	MailUserAgent
GUI	Graphical User Interface	RTS	Run-Time-System
		VM	VideoMail
		VP	VideoPlayer

10. Adaptors

Motivation

Split: One component uses several other components.
Dispatching of calls to the appropriate component.

Join: Serveral components use one (other) component.
Synchronizing calls to the simultaneously used component. simultaneously

Protocol Changers: Right functionality is offered, but within a different protocol.

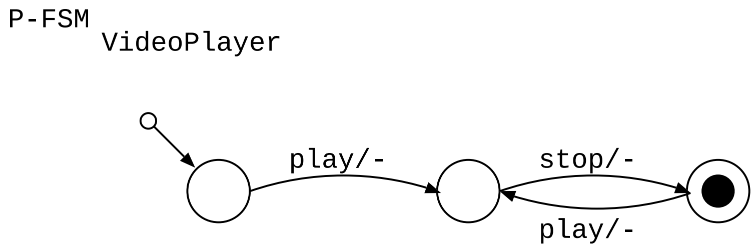
Adaption is not enough. Adaptors are necessary.

10. Split

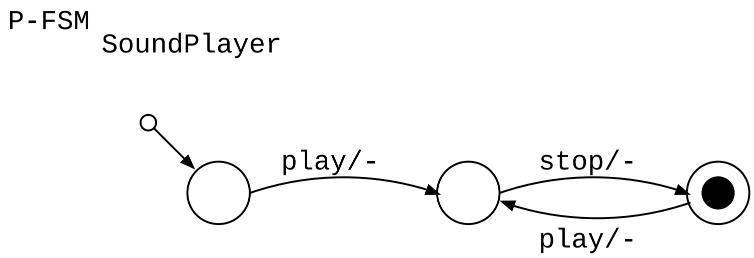
Example

`VideoPlayer` and `SoundPlayer` form a single Provides-Interface towards the `VideoMail`.

Provides-Interface of the `VideoPlayer`



Provides-Interface of the `SoundPlayer`

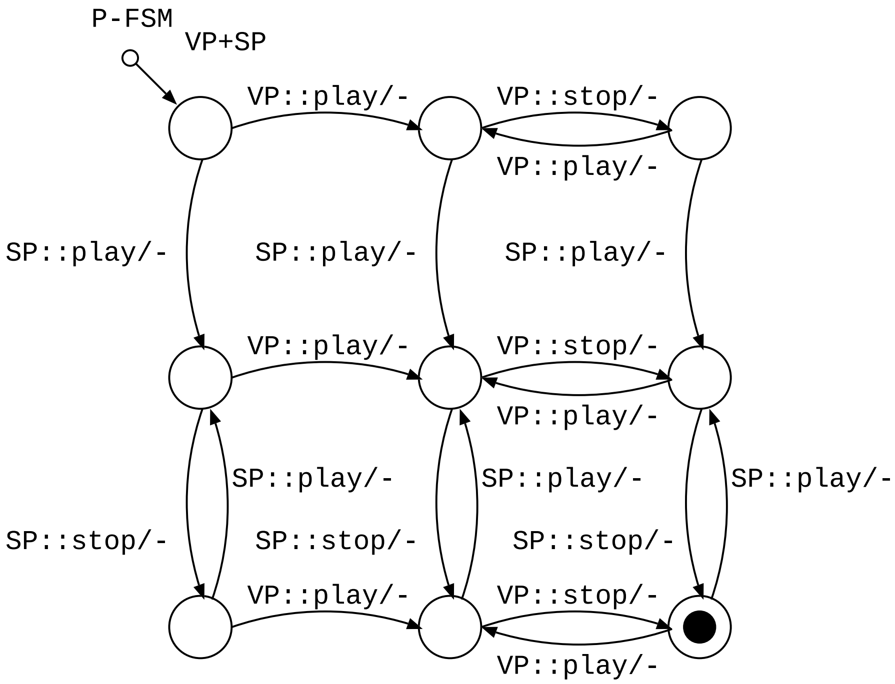


10. Split

Example

The resulting single interface of `VideoPlayer` and `SoundPlayer`: The Shuffle-FSM [Shaw 1978].

Shuffle-FSM as Provides-Interface of the `VideoPlayer` and `SoundPlayer`

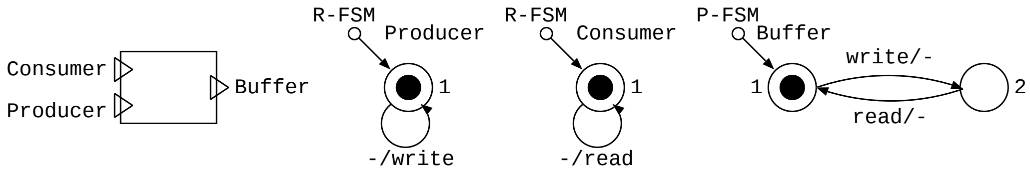


10. Join

Example

Inputs:

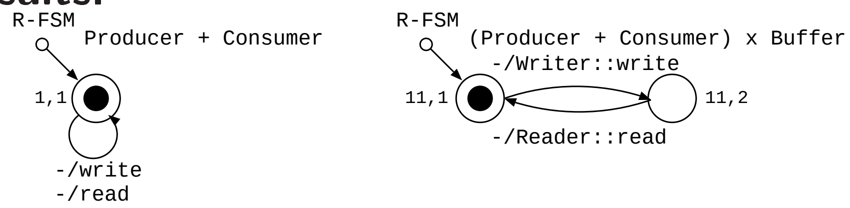
`Writer` as producer, `Reader` as Consumer and `Buffer` as input for the join-operator generation.



Algorithm:

1. Compute the shuffle-FSM $A + B$ out of the R-FSM$_A$ and R-FSM$_B$.
2. Build the intersection FSM of the shuffle FSM $A + B$ and P-FSM$_C$.
3. Derive synchronization information out of $(A + B) \times C$ and P-FSM$_A$

Results:



Result for the `CRI`$_{\texttt{Reader}}$`:read` operation

"enables CRI$_{\text{Reader}}$`:read_1`, enables CRI$_{\text{Writer}}$`:write_1`".

Result for the `CRI`$_{\texttt{Writer}}$`:write` operation

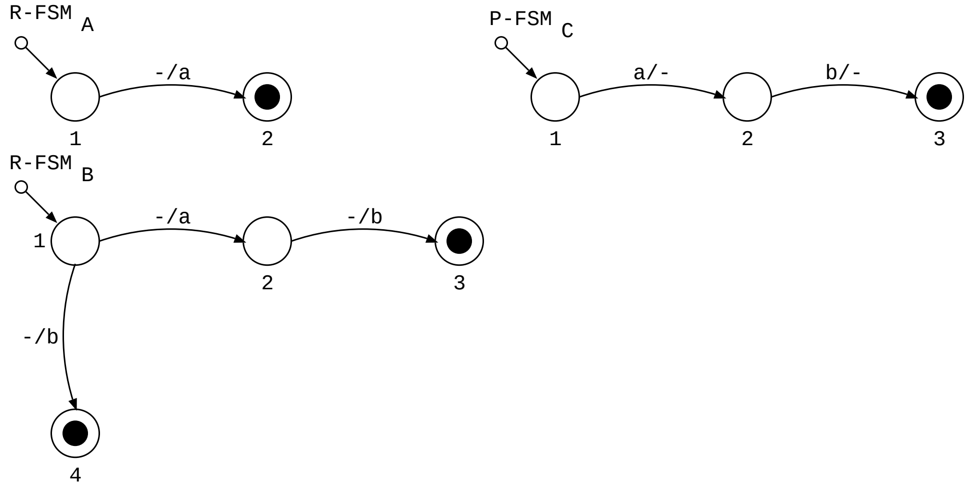
"enables CRI$_{\text{Writer}}$`:write_1`, enables CRI$_{\text{Reader}}$`:read_1`".

10. Join

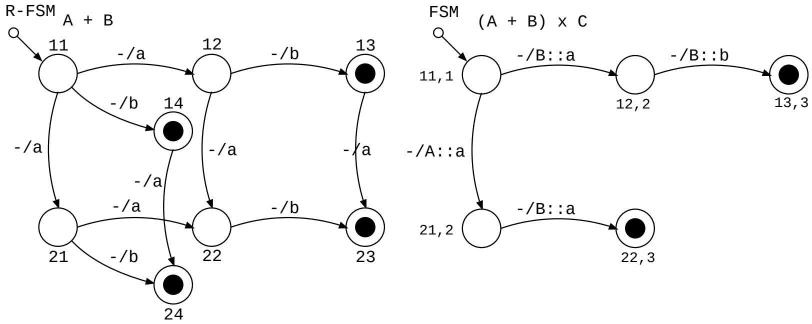
Second Example

Inputs:

P-FSM$_C$ as input for the join-operator generation.



Results:

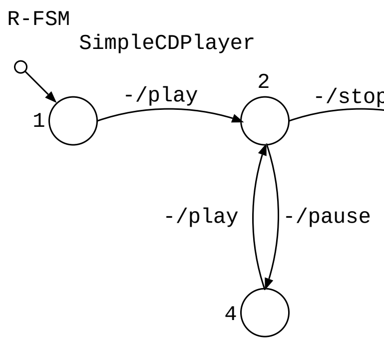


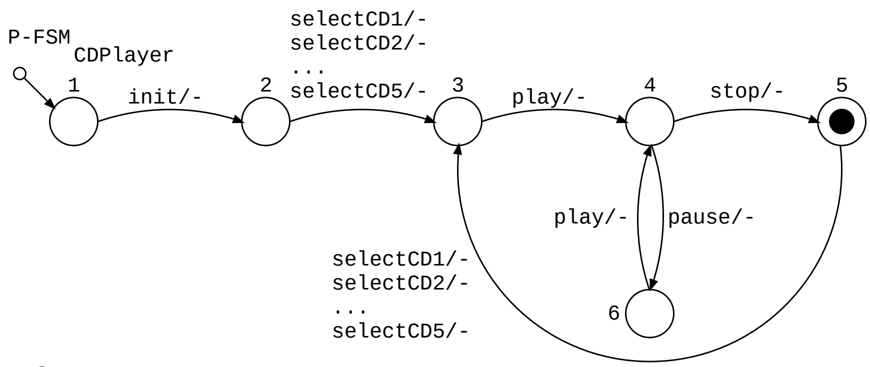
method call	annotations
A:a_1	excludes B:a_1,
	-,enables B:b_1
B:a_1	excludes A:a_1
B:b_1	enable B:b_1,-

10. Protocol Changing Adaptors

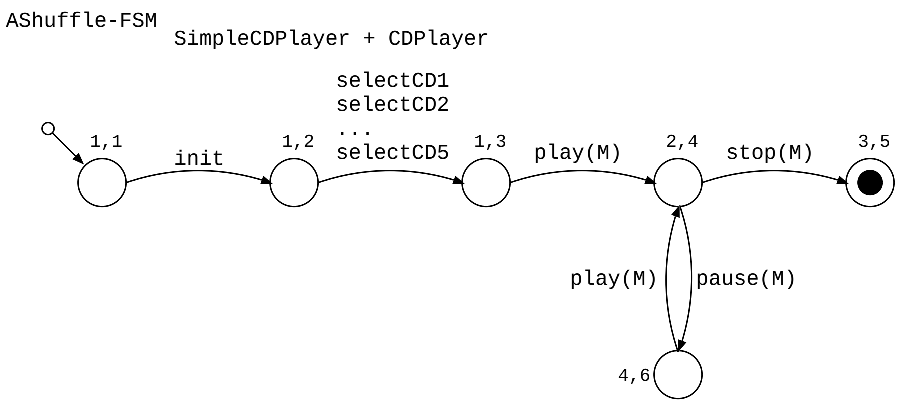
Example

Inputs:





Result:

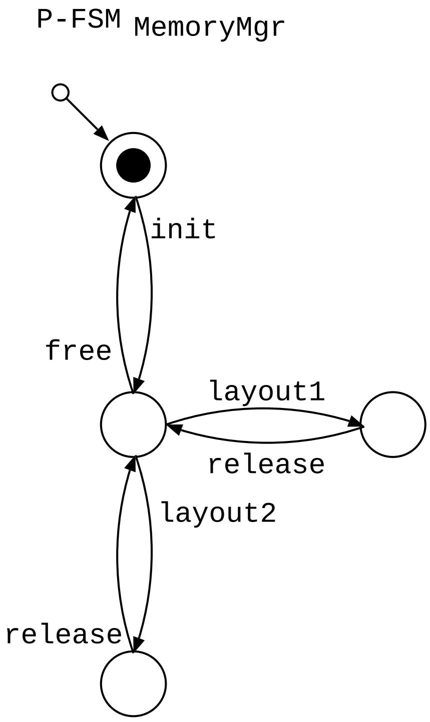


11. Hiding FSMs

Code-Oriented Approach

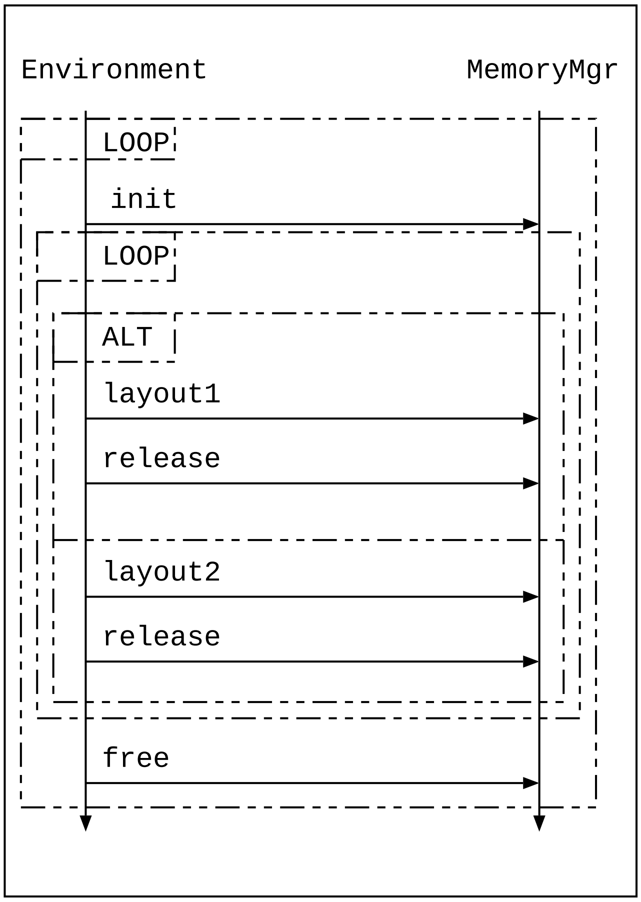
- Annotations (P-FSM)
- Analysis (MC-FSMs)
  - User's Security
  - Programmer's Convenience

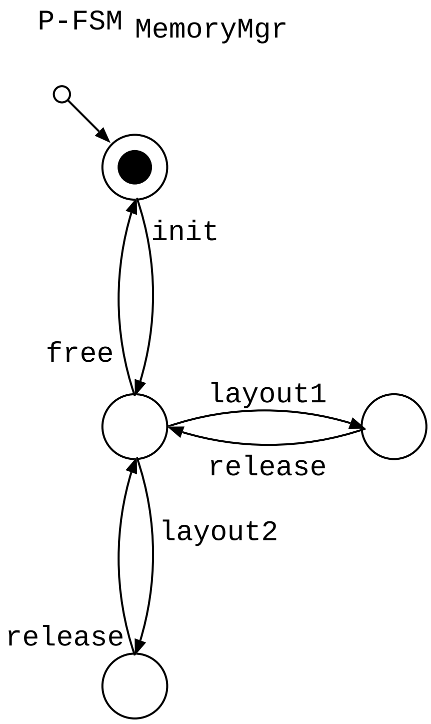
```
COMPONENT MemoryMgr {
  @nonConcurrent(layout1;release||
                 layout2;release||
                 free)

  @before free @before l1 @before l2
  @leadsTo free
  init { // request memory from OS
  ...}

  free { // restitute memory to OS
  ...}

  @leadsTo release
  layout1 { // memory layout 1
  ...}

  @leadsTo release
  layout2 { // memory layout 2
  ...}

  @after l1 OR l2
  release { // release memory layout
  ...}
}
```



Design-Oriented Approach
Generation of P-FSM out of MSCs

- Early Evaluation of Architecture
- Queries to Repository





12. Conclusions

- Problems of component-based software engineering
- Requirements to a type system for software components
- Parameterised Contracts
- Types for components
- Type adaption
- Type extension
- Meta-Protocol to control type changes
- Adaptors: Join, Split, Protocol Change
- Hiding FSMs: Annotations and MSCs

13. Future Work

- Integration into Architectural Description Language
- More elaborated dispatch mechanisms for type extension
- Synchronisation between components and adaptor
- Use of type information for user interfaces
- Non-functional security
- Integration of non-functional properties (timing, reliability, etc.)

14. Further Information

- URL: http://www.trusted-components.org
- URL: http://liinwww.ira.uka.de/~reussner/coconuts/coconuts.html

Appendix 1. Software-Components

An old thing?

McIllroy 1968: Idea of Software Components, Analogy to Hardware Components (later Cox's *Software-IC's*)

Parnas 1972: Guidelines to Modularization

Szyperski 1998: Software-Components as a

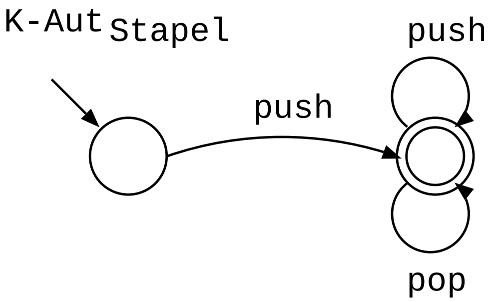
- independently deployable unit
- third party composition
- A component has no persistent state

D'Souza &Wills 1999: Changes by Adaption not by Modification.

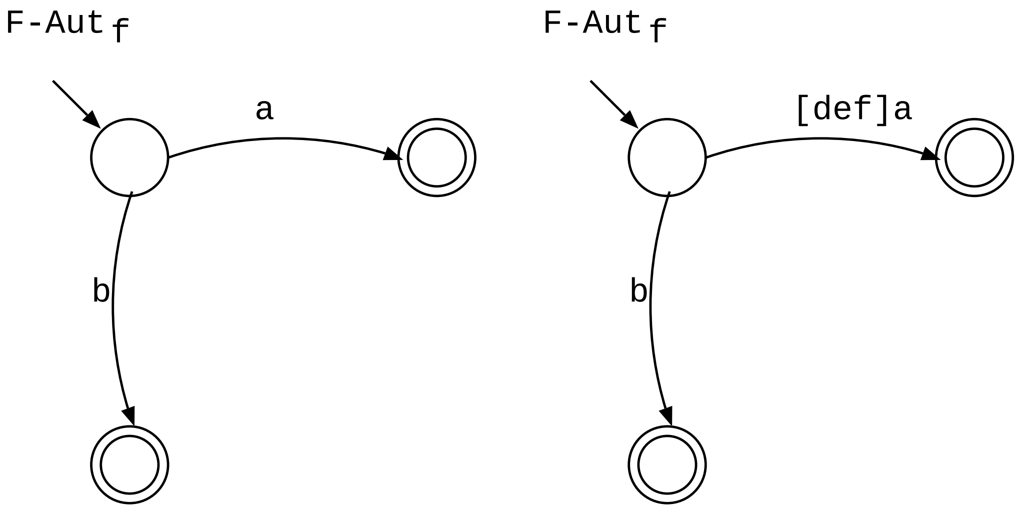
Goos 1999: Artefact plus deployment desription.

Why not FSMs?

- Not all protocols are describable with FSMs (e.g., Stack)⇒ Extension to counter-FSMs



- Extension for the "or-sematics"



Other Approaches

- Petri-Nets
  no normalized form, no intersection
  but extensible to describe FSMs.
- Nondeterministic FSMs [Nierstrasz 93]
  Large statespace $\Rightarrow$ high space- and time complexity of the algorithms.
- Deterministic push-down-FSMs (contextfree grammas)
  not completed under intersection.
- Linear limited FSMs (context sensitive grammas)
  Emptiness-, Intersection- and Equivalenceproblem not decidable.
- „State-affected" Interface: CDL [OMG 97]
  Description of the Sematics of a Component, only rudimentary protocols
- design contract [Meyer 89], [Zaremski & Wing 96]
  Complicated description of the call protocol. Calclulation of the new protocol difficult.