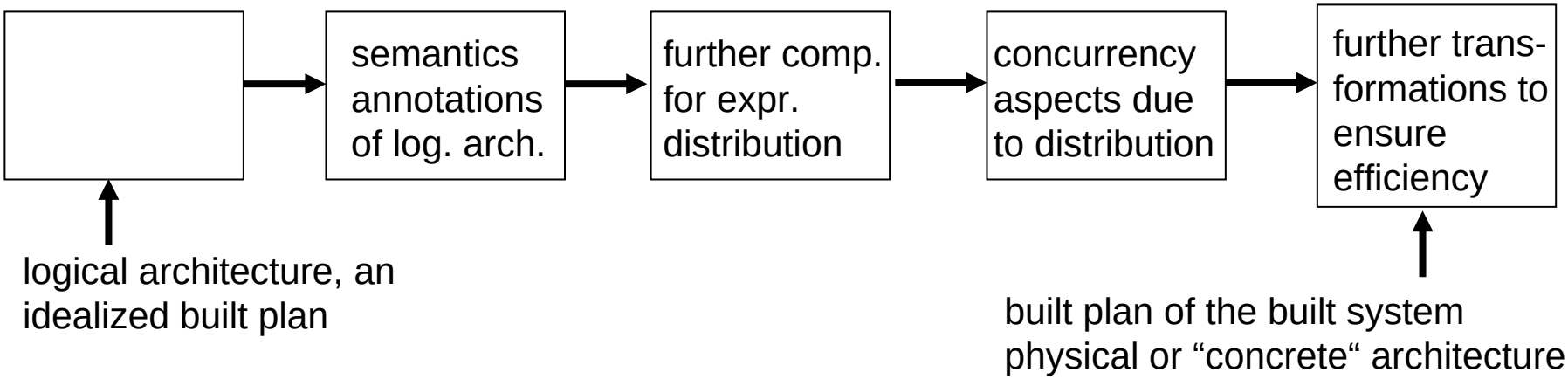
7. Extensions and Annotations

Aims

- Further important aspects yielding to extensions of / annotations to architectural languages:
  not further isolated view but details of ex. architecture
- Aspects:
  Semantics
  Concurrency
  Distribution
- There is not one architecture but a series, for example



Contents of Chapter 7

... to be elaborated:

nonfunctional properties of systems

efficiency transformations

concurrency properties

commonalities / differences of program families

etc.

Semantics of Components (Example adt module)

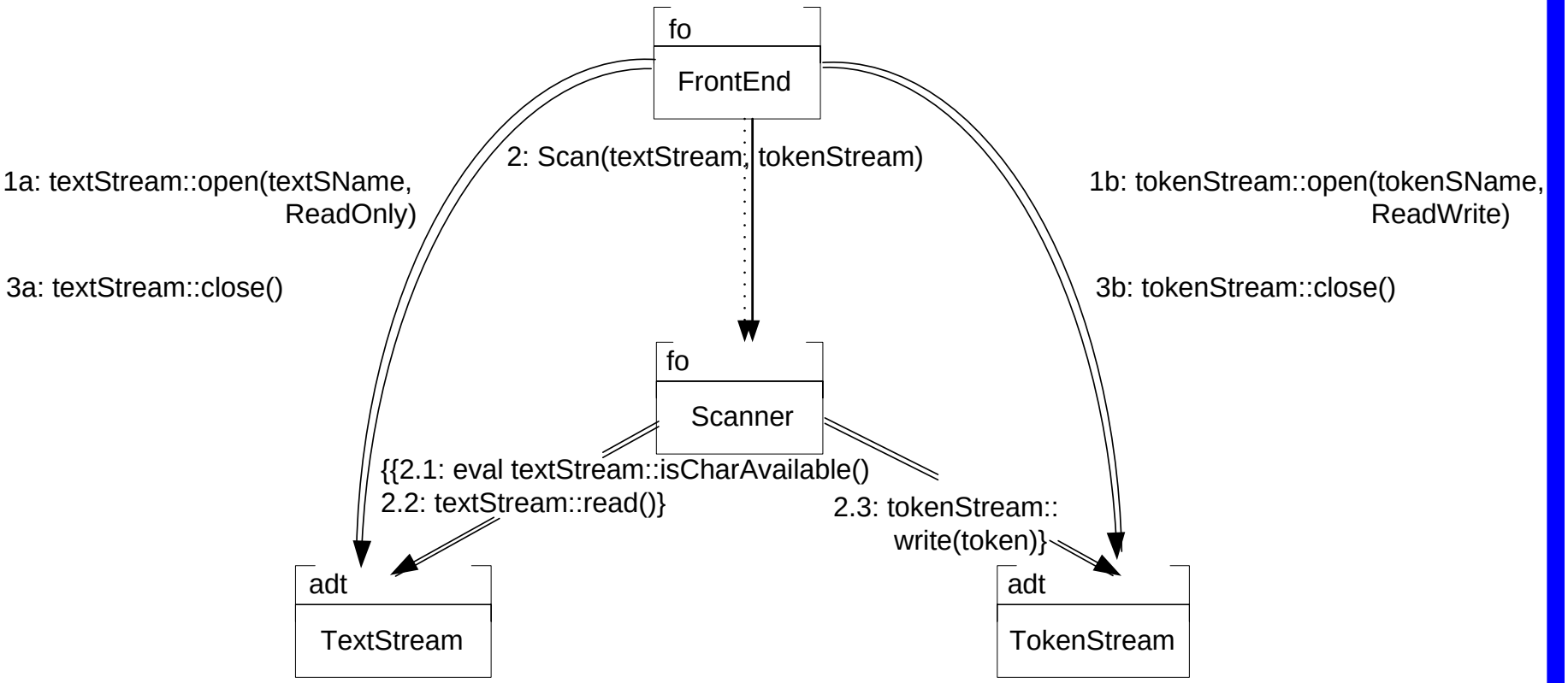
```
spec ITEM_STACK_TYPE
    -- imports of other specs are necessary, e.g. of ITEM
    sorts ITEM_STACK_TYPE
    operations
                    -- export interface:
                    NEW: ———> ITEM_STACK_TYPE
                    PUSH: ITEM x ITEM_STACK_TYPE |———> ITEM_STACK_TYPE
                    POP: ITEM_STACK_TYPE |———> ITEM_STACK_TYPE
                    READ_TOP: ITEM_STACK_TYPE |———> ITEM
                    IS_EMPTY: ITEM_STACK_TYPE ———> BOOLEAN
                    IS_FULL: ITEM_STACK_TYPE ———> BOOLEAN
```

signature: sort / ops part; partial / total functions

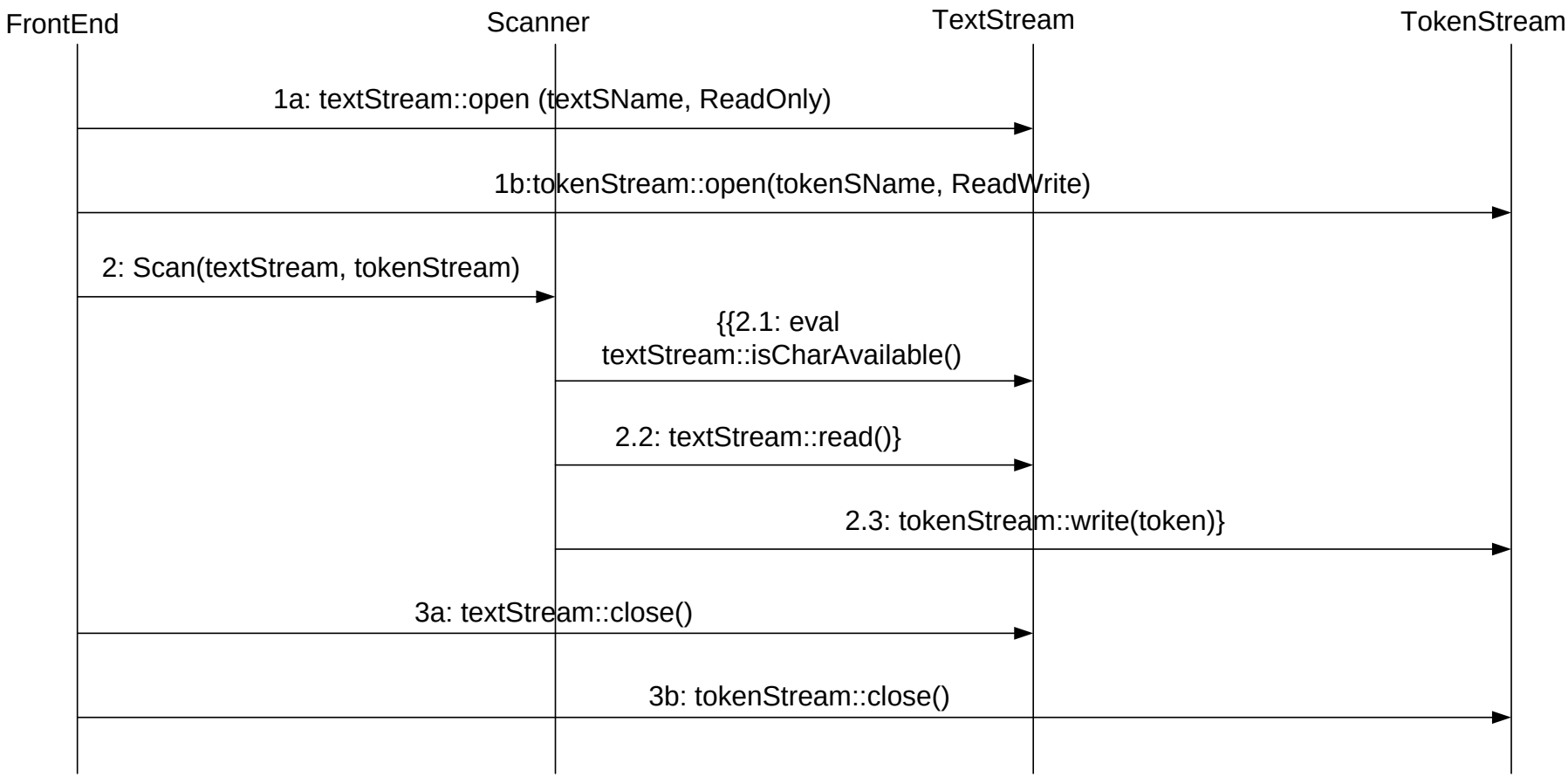
```
    preconditions
                    -- preconditions of ops such that alg. spec. holds
                    pre POP(ST: ITEM_STACK_TYPE) = not IS_EMPTY(ST)
                    pre READ_TOP(ST: ITEM_STACK_TYPE) = not IS_EMPTY(ST)
                    pre PUSH(EL: ITEM; ST: ITEM_STACK_TYPE) = not IS_FULL(ST)
    equations
                    for all EL: ITEM, ST: ITEM_STACK_TYPE:
                    IS_EMPTY(NEW())
                    not IS_EMPTY(PUSH(EL, ST))
                    not IS_FULL(NEW())
                    not IS_FULL(POP(ST))
                    READ_TOP(PUSH(EL,ST)) = EL
                    POP(PUSH(EL, ST) = ST
end spec ITEM_STACK_TYPE
```

semantics of ops by determining mutual relations, assuming preconditions

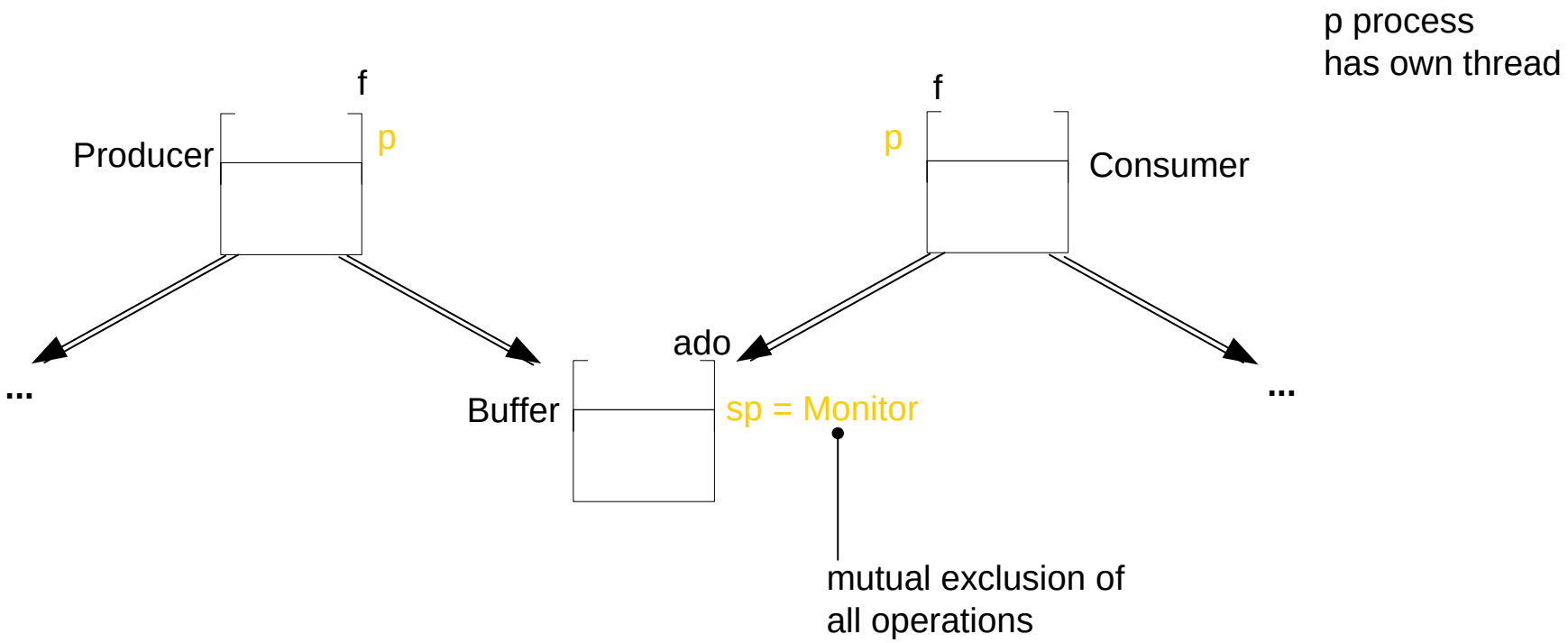
Specifying Component Interaction by Traces



Alternative Sequence Diagram ?



Annotations for Concurrency



derived “process description“ = synchronization protocol of a data structure

other protocols:

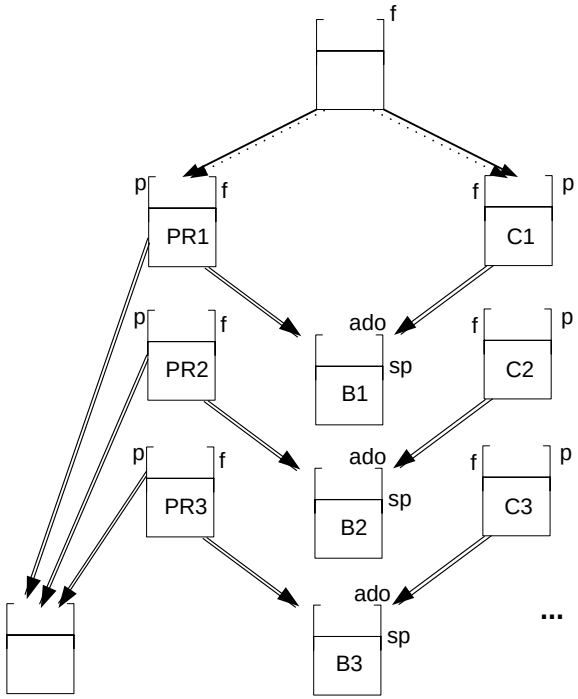
protected objects / types of Ada 95:

parallel readers if no writer

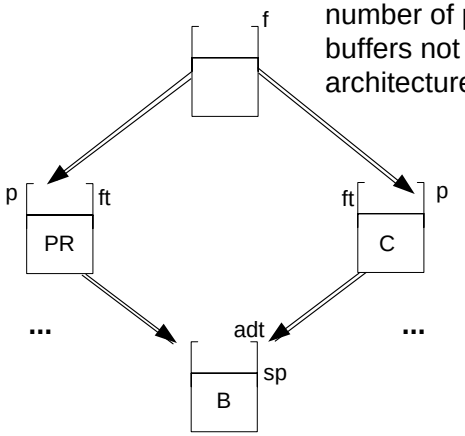
etc.

Necessity of ft Modules

- More than one producer / consumer of same behaviour
- f modules (objects) by replication or ft modules ? } good reason for both solutions
- ft modules, if number of producers / consumers (not) known at development time



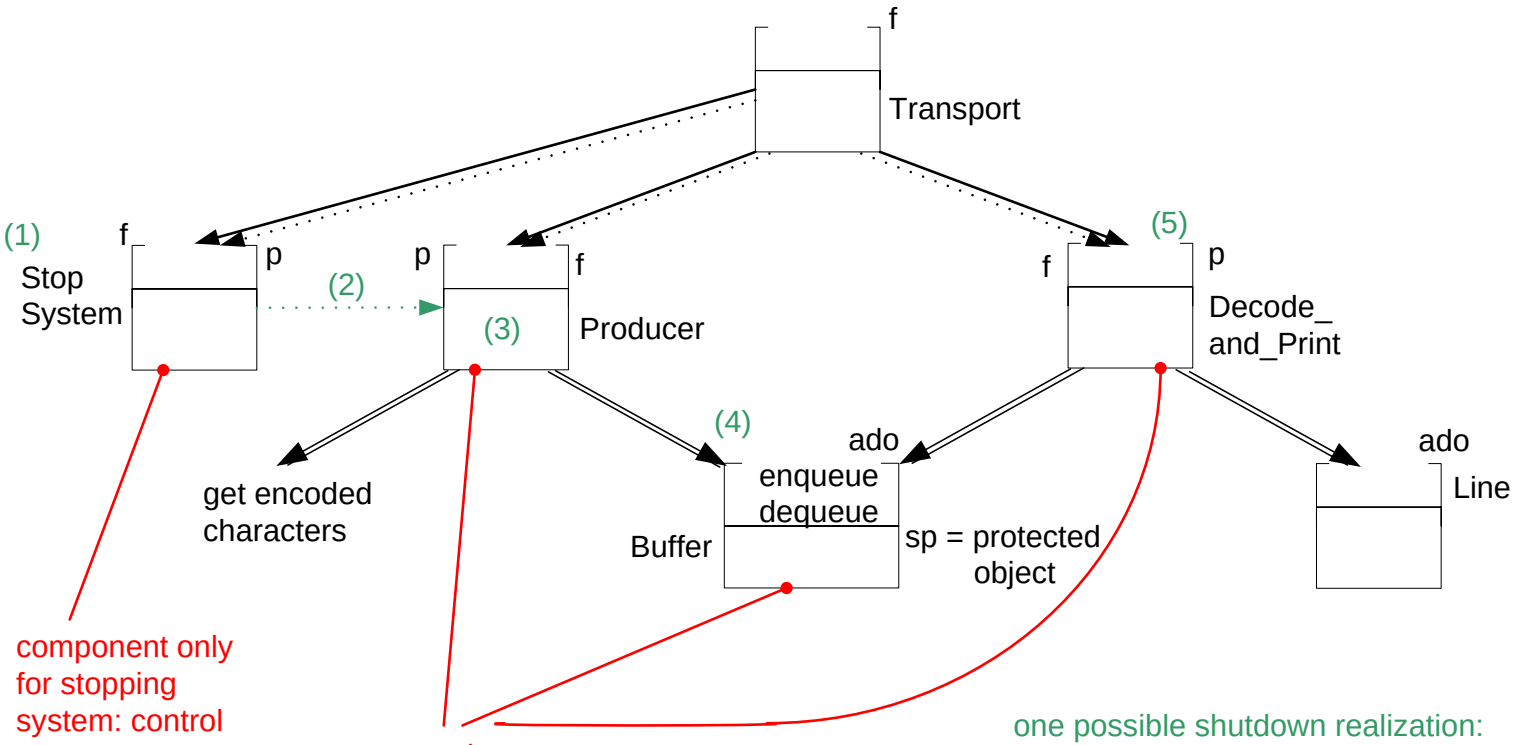
solution with producer / consumer objects



number of producer / consumer buffers not to be seen in architecture

solution with producer / consumer types

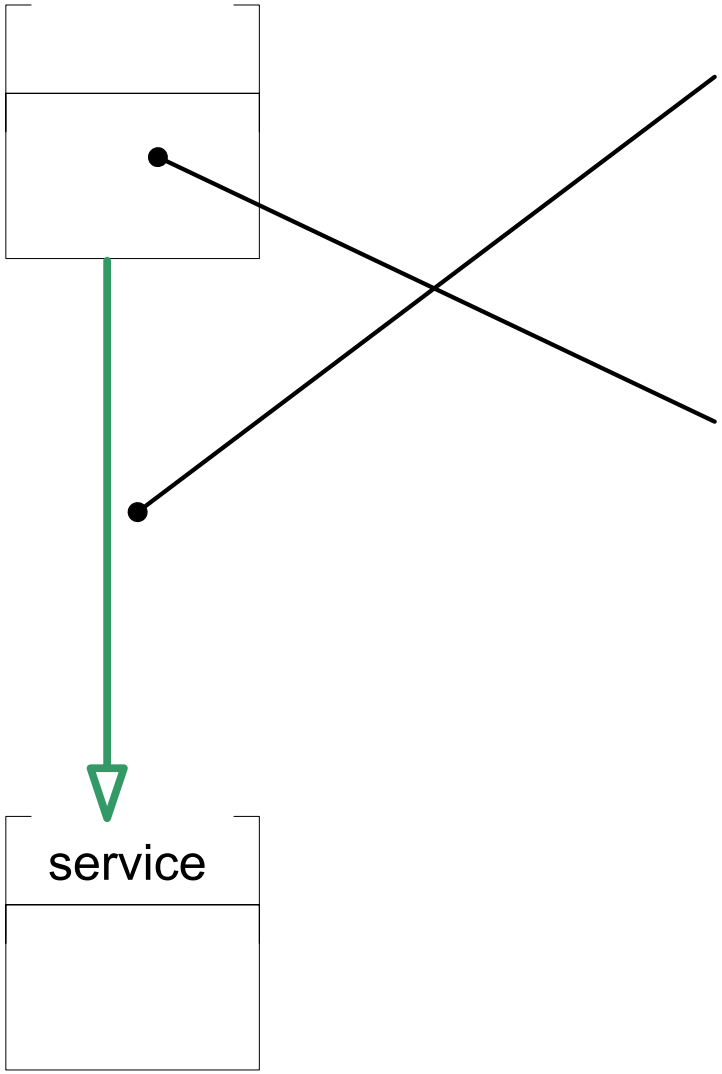
Denoting Start / End in Embedded Systems



one possible shutdown realization:

(1) Press Button: entry call
(2) Stop Entry of Producer is activated
(3) Producer stops working
(4) Stop sign put on Buffer
(5) Decoder stops himself when reading this sign

Usability as a Logical Concept



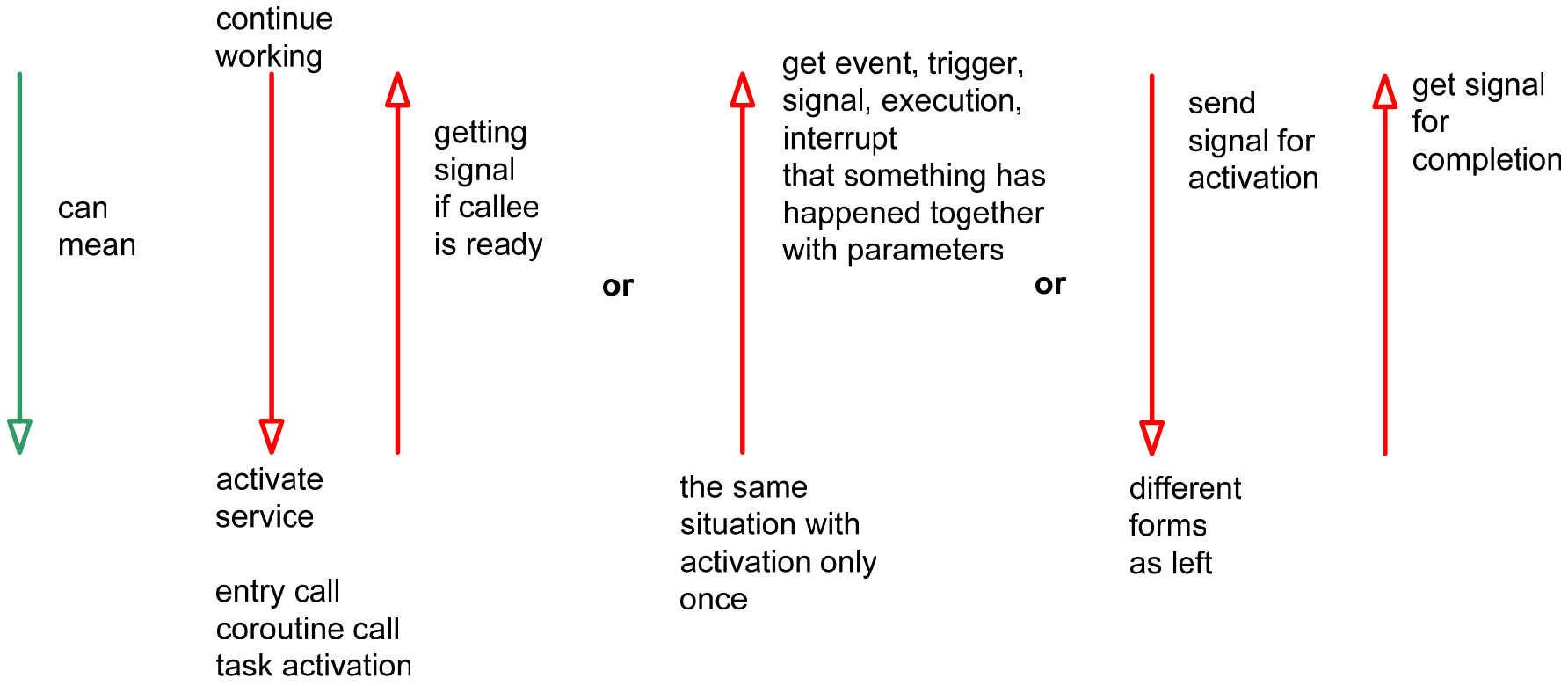
either local,
general,
inheritance
} import

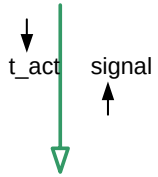
body uses a service of next interface (or interface uses a type of next interface)

that up to now means to call a function / procedure
being the service of a f, ado, adt module

with “calls“ we up to now combine that caller waits until callee has finished and then proceeds

Different Concrete Forms of Usability

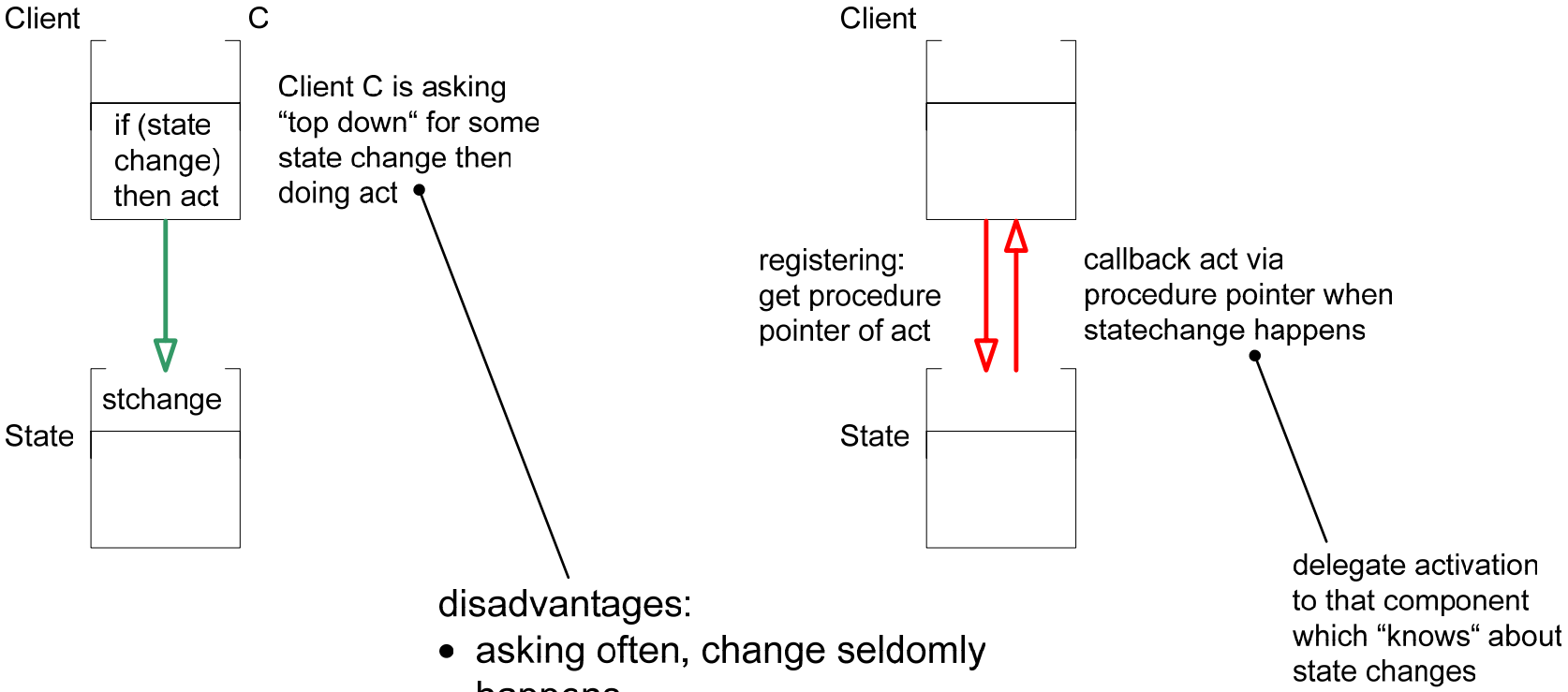




⇨ Different forms of concrete usability

Make annotations at usability edges, e.g.

“Reverse“ Use by Callbacks



disadvantages:

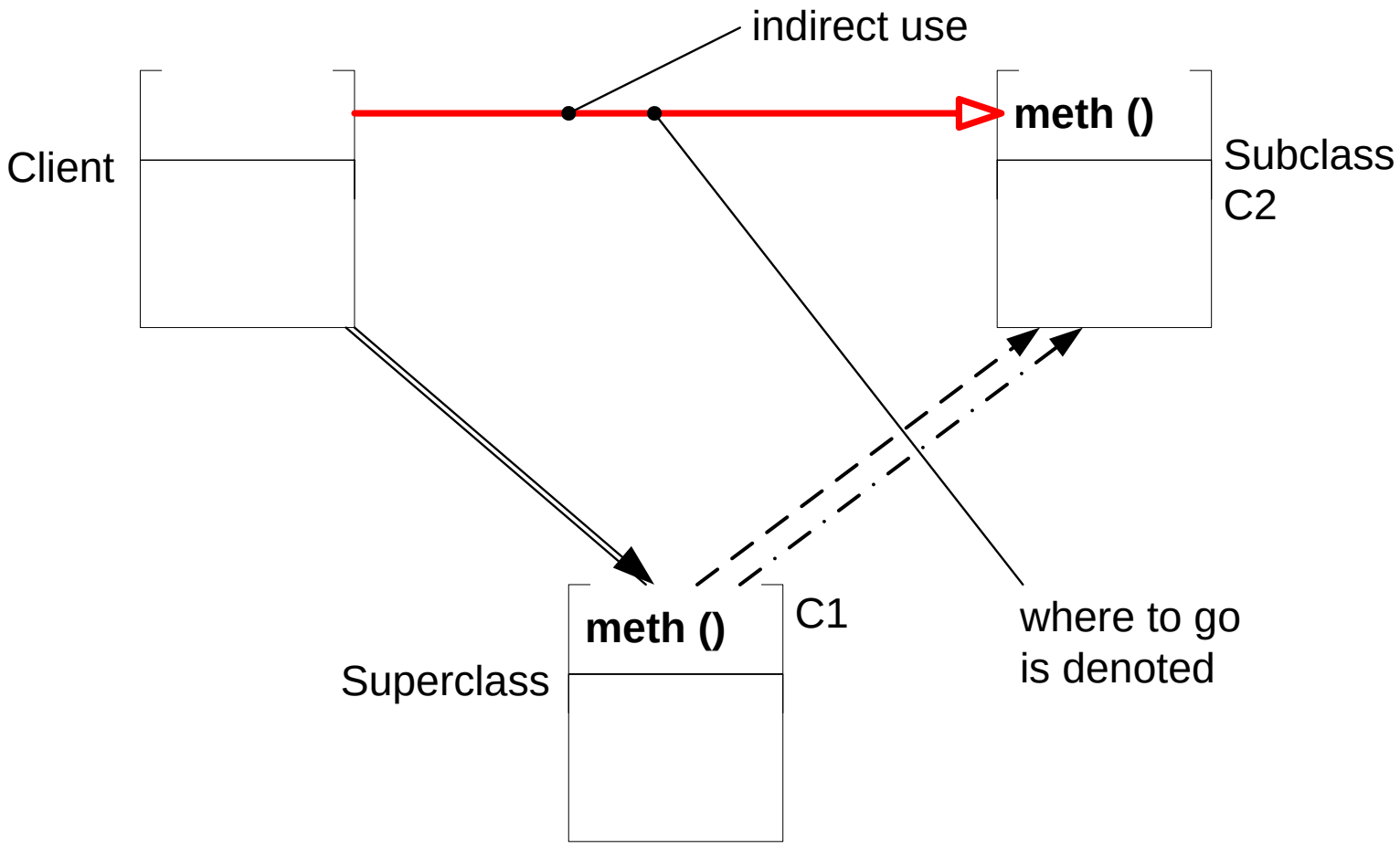
- asking often, change seldomly happens
- if state is also changed by other clients: don´t know when

Applications:

GUI User has clicked: Bind Application function

Active Databases: Bind change or repair action

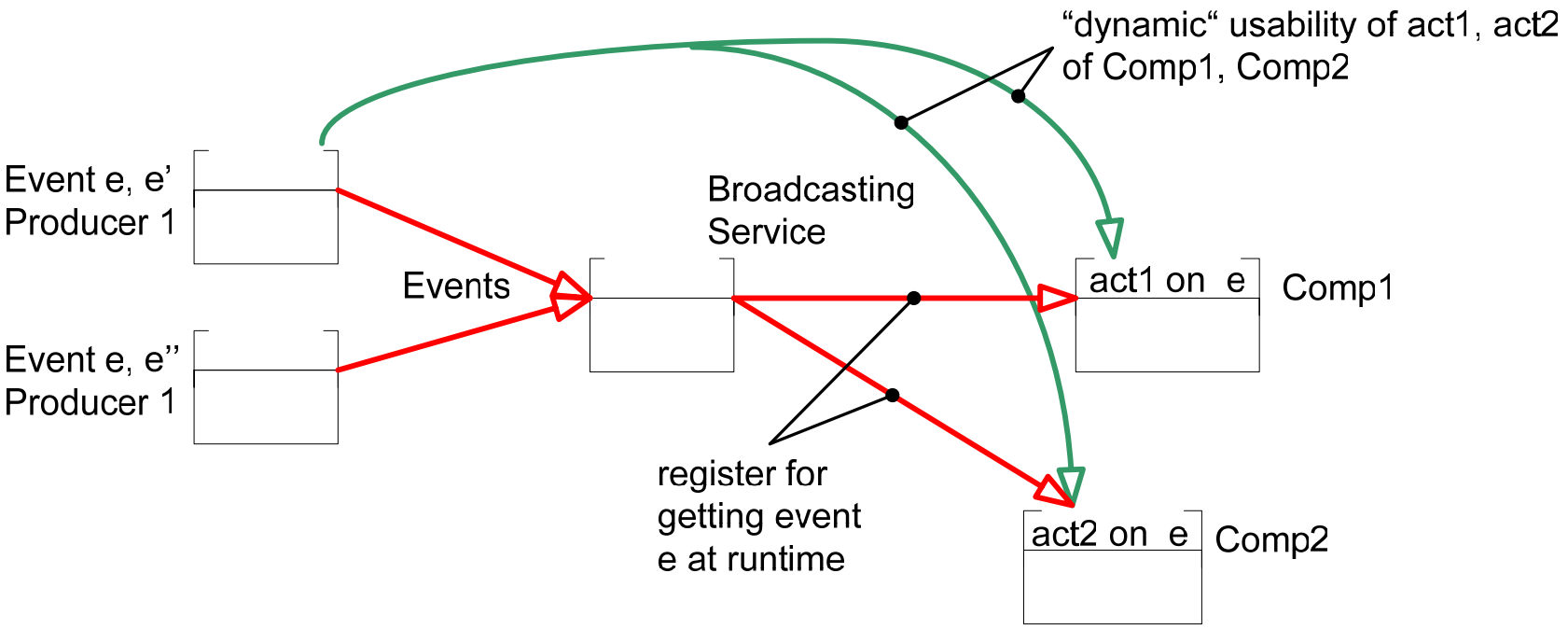
“Indirect“ Use by Dispatching



client calls superclass, e.g. when handling objects of an inheritance hierarchy

superclass “calls“ subclasses via dispatching

Dynamic Use via Broadcasting / Registration Service



“Dynamic“ use should be planned at design time

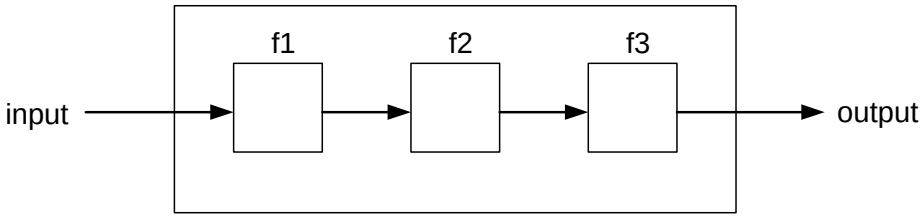
Only dynamic uses corresponding to planned ones ?

Analogously when building pure event-based systems

Specific Application-oriented Notations

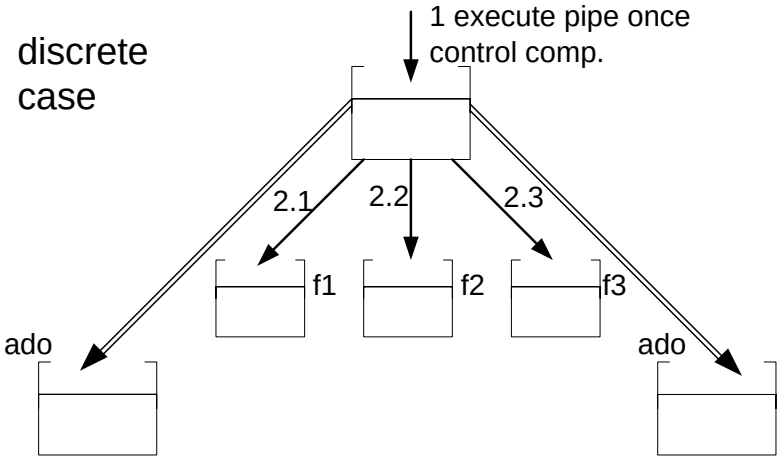
- Show extensions of our Architecture Languages
  thereby restrict on graphical form
- Show Mapping on our Architecture Languages
  ⇨ extensions are used as abbreviations
- Styles:
  - Pipelines
  - Data Flow Architectures
  - Loosely Coupled Systems

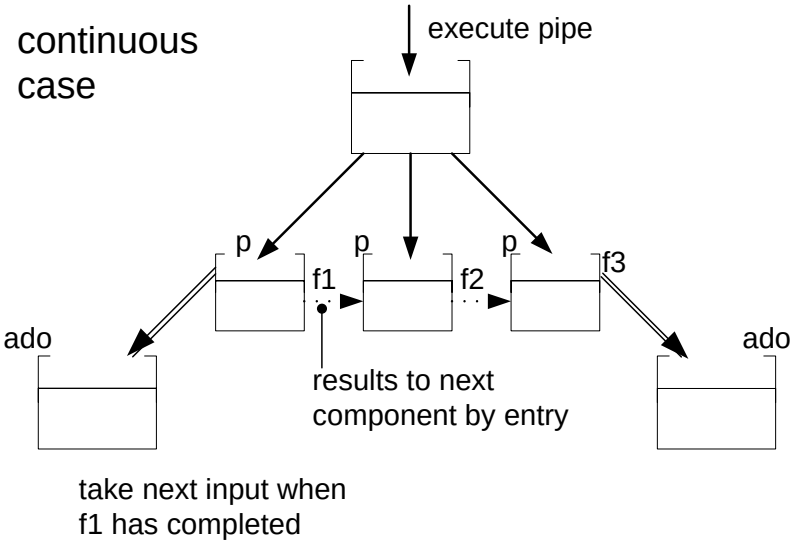
Pipelining Architectures



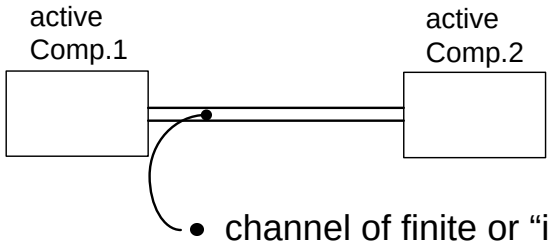
the result of f1 is input to f2; the result of f2 ist input to f3; ...

discrete calculation / continuous calculation





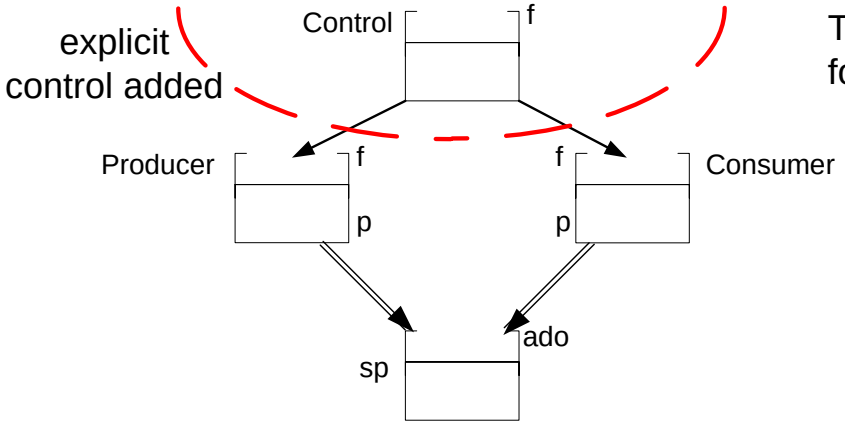
Data Flow-Oriented Interaction



data flow-oriented architectures
are used in embedded systems
have usually “cycles“

- channel of finite or “infinite“ capacity
- general signal, or distinction between data or control signals
- different semantics for production / consumption:
  a new signal destroying an old, a consumed is deleted / not,
  continuous / discrete signals due to production / consumption,
  and / or for incoming / outcoming data flows

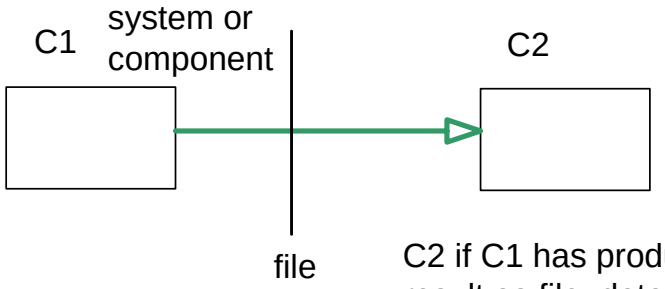
Such a notation can be regarded as abbreviation of our notation



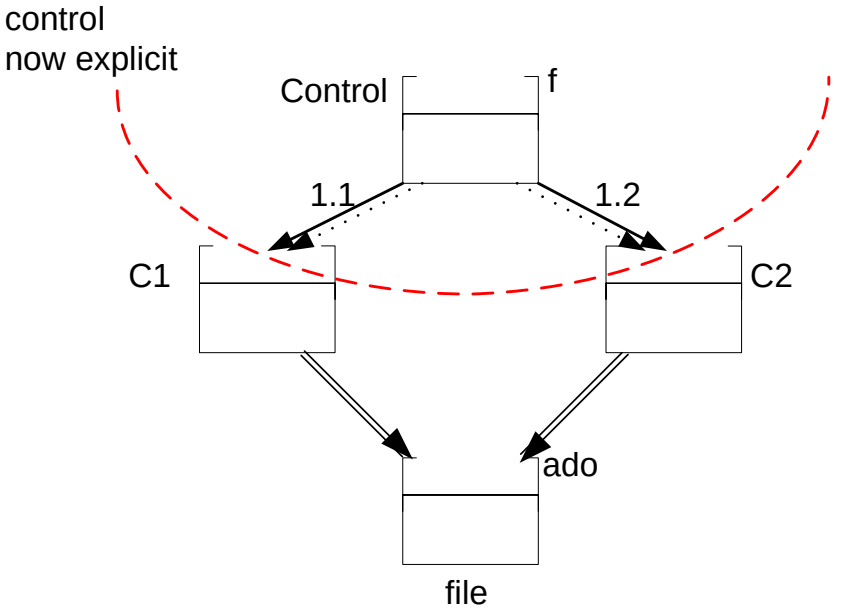
This allows also to
formulate cycles

⇨ new forms of diagrams to be translated to usual architecture diagrams

Indirection via Loose Data Coupling



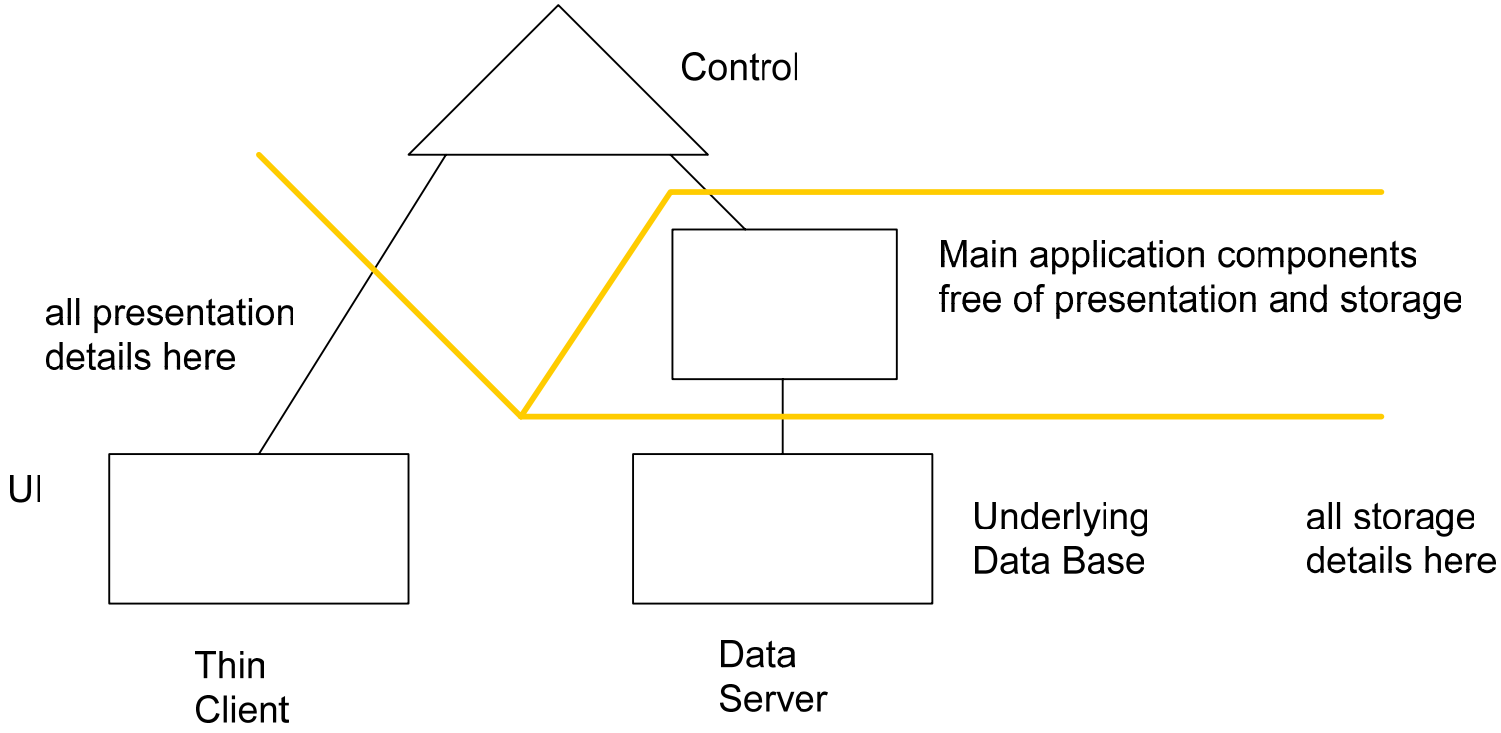
C2 if C1 has produced result as file, database etc.,
then C2 is started



coupling two
sequential systems

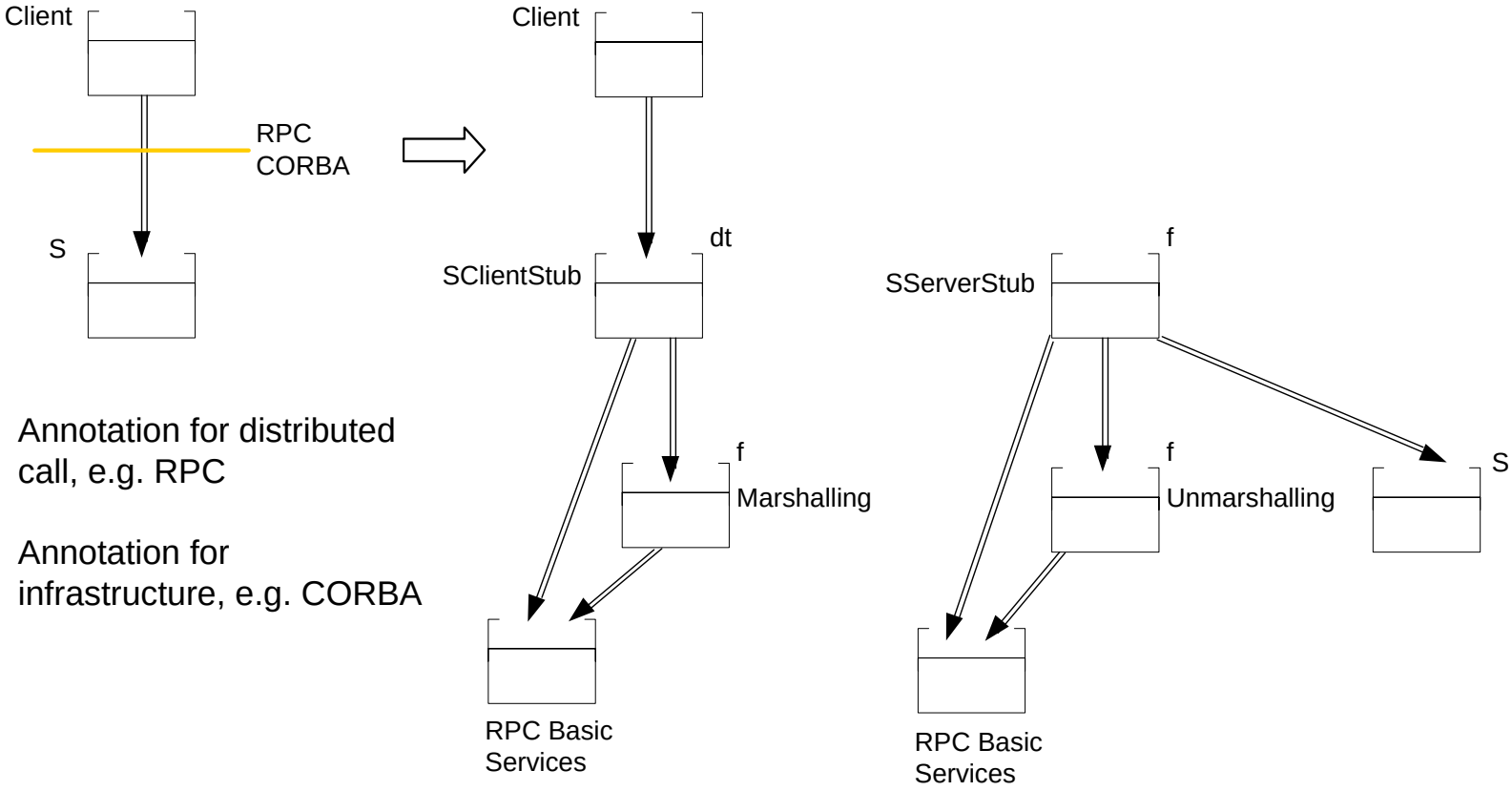
Suitable Distribution Lines

- Where to cut ? Characterization:
  - » clear areas of concern
  - » loose logical coupling: small interface, little knowledge of the other side
  - » low communication rate
- Example for interactive information systems



Distribution and Communication: Further Components

Distribution Mechanism: RPC as example



Annotation for distributed call, e.g. RPC

Annotation for infrastructure, e.g. CORBA

client – side components

server – side components

Distribution and Induced Concurrency