Distributed System Design from Global Requirements

Gregor v. Bochmann

School of Electrical Engineering and Computer Science (EECS)
University of Ottawa
Canada

uOttawa

L'Université canadienne
Canada's university

http://www.site.uottawa.ca/~bochmann/talks/Deriving.ppt

Plenary talk at CCECE 2012

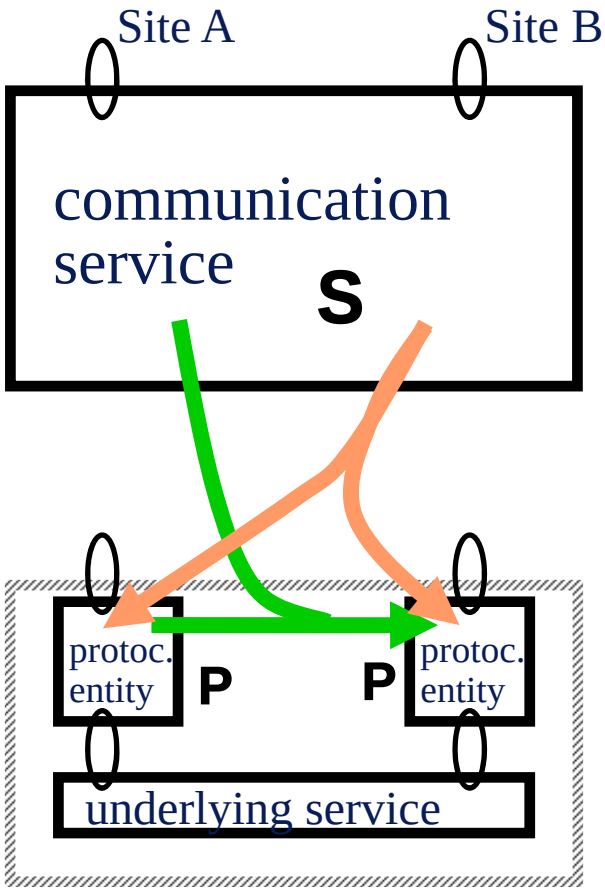
Montréal, May 1st, 2012

Abstract

Distributed systems are difficult to design because (1) message exchanges between the different system components must be foreseen in order to coordinate the actions at the different locations, and (2) the varying speed of execution of the different system components, and the varying speed of message transmission through the different networks through which the components are connected make it very hard to predict in which order these messages could be received. This presentation deals with the development of distributed applications, such as communication systems, service compositions or workflow applications. It is assumed that first a global requirements model is established that makes abstraction from the physical distribution of the different system functions. Once the architectural (distributed) structure of the system has been selected, this global requirement model must be transformed into a set of local behavior models, one for each of the components involved. Each local behavior model is then implemented on a separate device, and realizes part of the system functions. It includes local actions and the exchange of messages necessary to coordinate the overall system behavior.

The presentation will first review several methods for describing global requirements and local component behaviors, such as state machines, activity diagrams, Petri nets, BPEL, sequence diagrams, etc. Then a new description paradigm based on the concept of collaborations will be presented, together with some examples. The second part of the presentation will explain how local component behaviors can be derived automatically from a given global requirements model. Also the implementation of these behaviors using BPEL software environments for Web Services will be discussed. Finally, some novel approach to testing behaviors defined as collaborations will be presented and an outlook at possible applications in the context of service compositions, workflow modeling, Web Services and Cloud Computing will be discussed.

Historical notes (some of my papers)



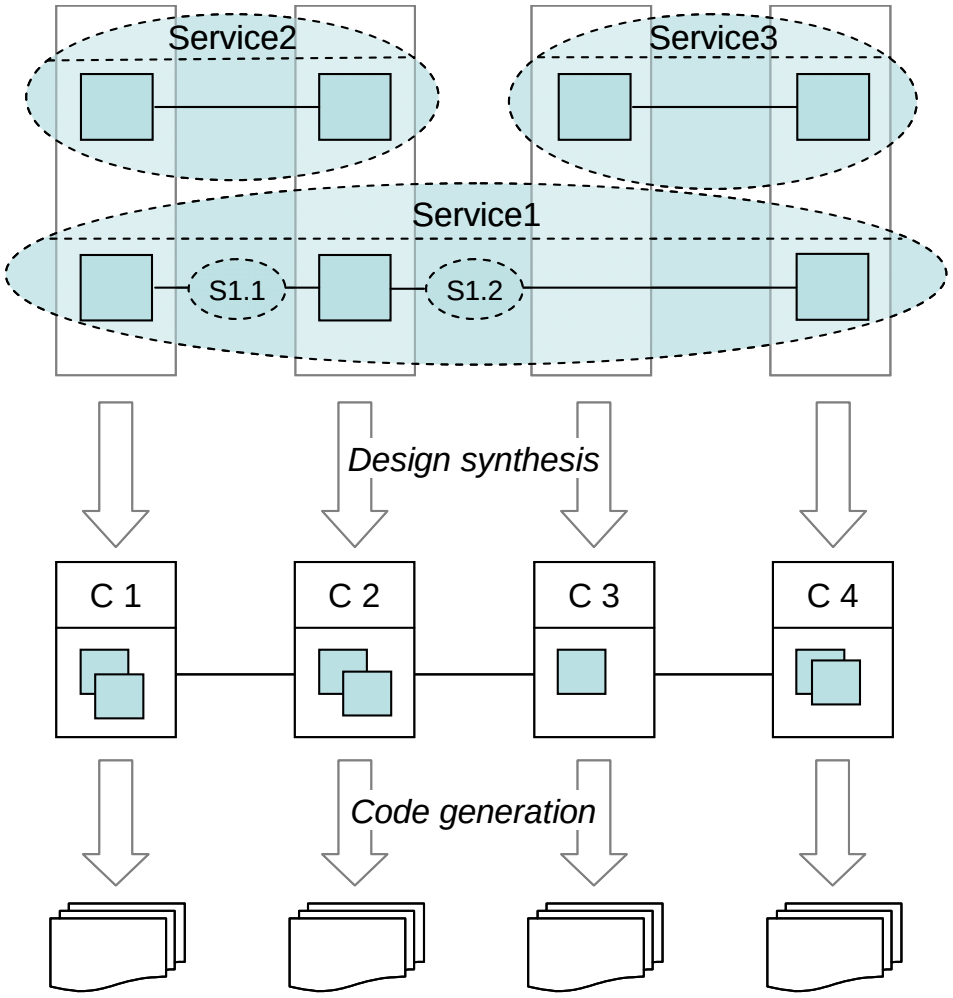
- 1978: meaning of

"a protocol P provides a service S"
(Finite State Description of Communication Protocols)

- 1980: submodule construction (with Philip Merlin)
- 1986: protocol derivation (with Reinhard Gotzhein)
- 2006: service modeling with collaborations (with Rolv Braek and Humberto Castejon)

Site A Site B

The problem – a figure



Service models

Design models

Implementations

Type of applications

- Communication services
  - telephony features (e.g. call waiting)
  - teleconference involving many parties
  - Social networking
- Workflows
  - Intra-organization, e.g. banking application, manufacturing
  - inter-organisations, e.g. supply-chain management
  - Different underlying technologies:
    - Web Services
    - GRID computing - Cloud computing
    - multi-core architectures
  - Dynamic partner selection: negotiation of QoS – possibly involving several exchanges

The problem

(early phase of the software development process)

- Define
  - Global functional requirements
  - Non-functional requirements
- Make high-level architectural choices
  - Identify system components
  - Define underlying communication service
- Define behavior of system components:
  - Locally performed functions
  - Communication protocol
    - Required messages to be exchanged and order of exchanges
    - Coding of message types and parameters

Issues

- Define
  - Global functional requirements
  - Non-functional requirements
- Make high-level architectural choices
  - Identify system components
  - Define underlying communication service
- Define behavior of system components
  - Local functions
  - Protocol:
    - Required messages to be exchanged and order of exchanges
    - Coding of message types and parameters

What language / notation to use for defining global requirements (dynamic behavior)

Architectural choices have strong impact on performance

Automatic derivation of component behaviors ? e.g. [Bochmann 2008]

Performance prediction – based on component behavior

- Response time, Throughput, Reliability

Choice of middleware platform for inter-process communication

- E.g. Java RMI, Web Services, etc.

Different system architectures

- Distributed architectures
  - Advantages: concurrency, failure resilience, scalability
  - Difficulties: communication delays, coordination difficulties
- Distribution-concurrency at different levels:
  - Several organizations
  - Different types of computers (e.g. servers, desk-tops, hand-held devices, etc.)
  - Several CPUs in multi-core computers

Proposed notations for global requirements

- UML Sequence diagrams
- UML Activity diagrams
- UML hierarchical State diagrams
- Use Case Maps
- XPDL (workflow) - BPMN (business process)
- BPEL (Web Services) – Note: defines centralized behavior
- WS-CDL ("choreography")
- Collaborations – a variant of Activity diagrams (joint work with university of Trondheim, Norway) [Castejon 2011]

Question:

How do they fit with the above issues ?

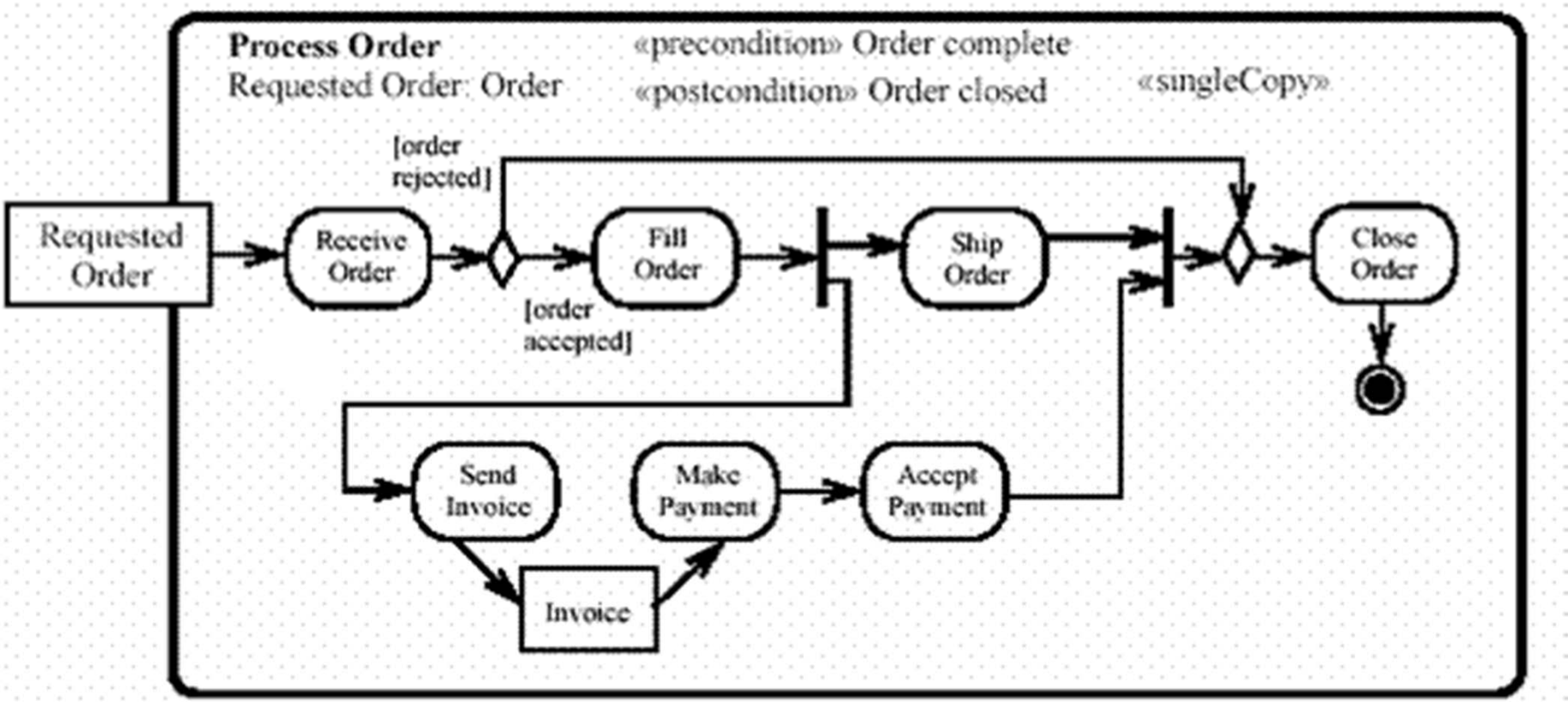
Overview of this talk

1. Introduction
2. Formalisms for describing global dynamic behaviors
3. Deriving component behaviors
   - 3.1 Distributed workflows
   - 3.2 Strong sequencing between sub-collaborations
   - 3.3 Weak sequencing between sub-collaborations
   - 3.4 Summary
4. Conclusions

2. Describing functional requirements

- The functional requirements are usually defined through a number of use cases.
- Use cases may be complex and need to be defined precisely.
- We consider the following notations for this purpose
  - For structural aspects:
    - UML Component diagrams
    - UML Collaboration diagrams
  - For the dynamic behavior:
    - dynamic collaboration diagrams - a variant of UML Activity or State diagrams (formalization: Petri nets)
    - Sequence diagrams (only for simple cases)

Example of an Activity Diagram



Concepts

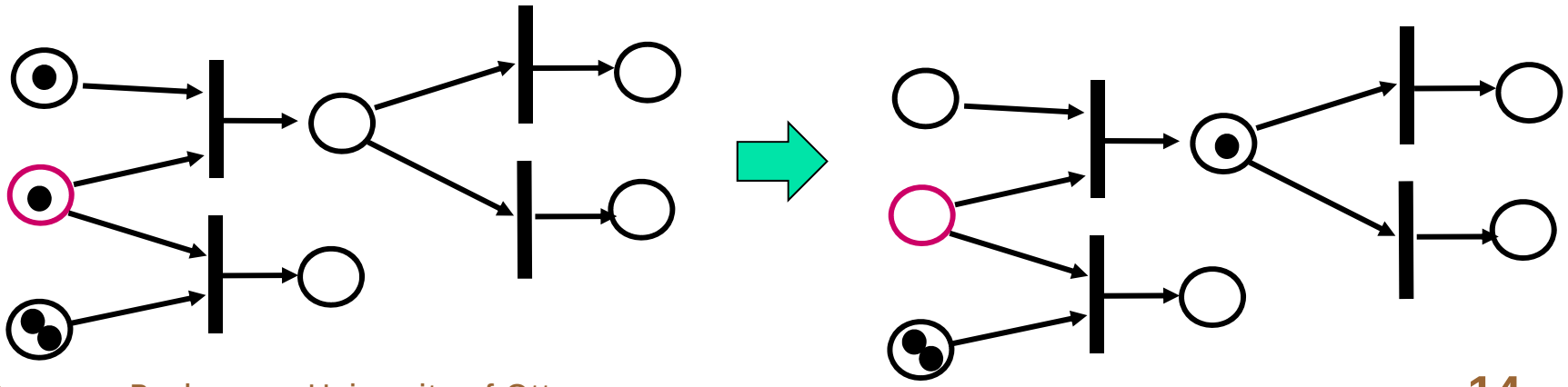
- Each Use Case is a scenario
  - Actions (Activities) done by actors in some given order
  - Actor: Swimlane - we call it component or role
  - Order of execution:
    - sequence, alternatives, concurrency, arbitrary control flows (can be modeled by Petri nets)
    - Interruption through priority events (not modeled by Petri nets)
- Abstraction: refinement of activity
- Data-Flow: Object flow - Question: what type of data is exchanged (an extension of control flow)
  - Input assertions for input data flow
  - Output assertions for output data flow
  - Conditions for alternatives

Petri nets

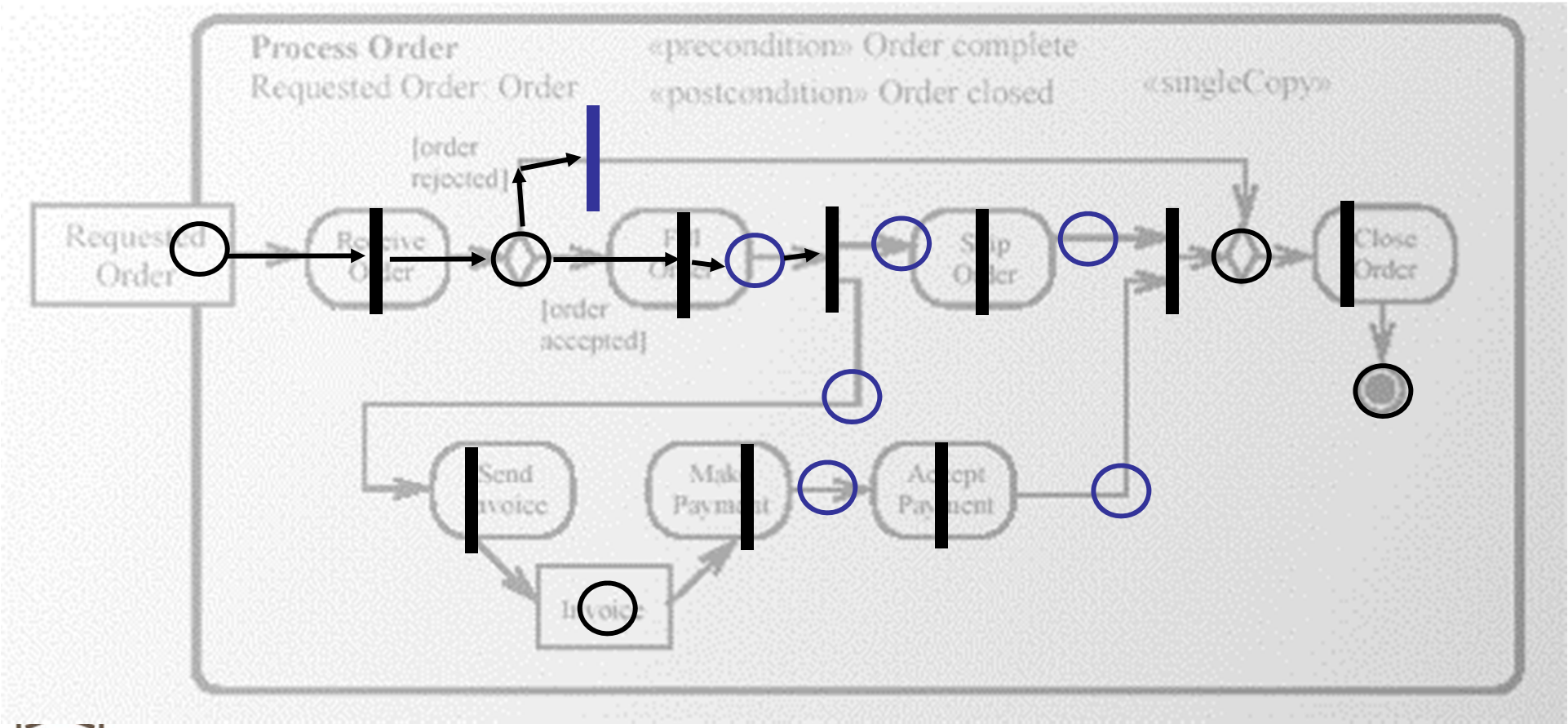
Defined by Petri in 1960. A net contains

- places: ◯ , may hold tokens
- Transitions: ▮ , represent *actions* that consume tokens from their input places and produce tokens for their output places.
- Tokens may contain data.

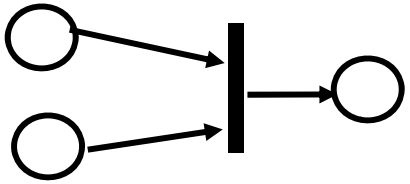
This diagram shows what happens when one transition is executed (fired):



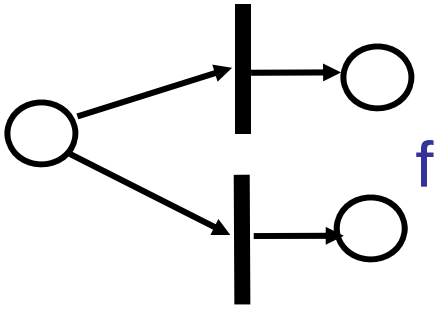
Activity Diagram: the corresponding Petri net



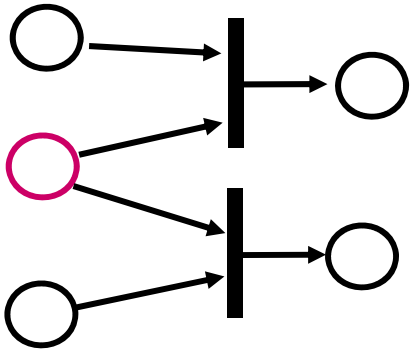
Free-choice nets – local choice



no choice



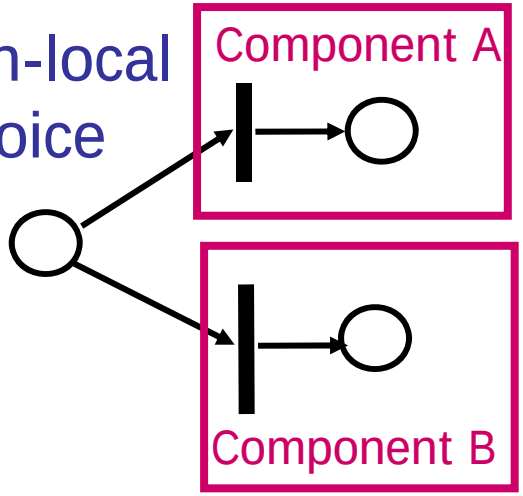
free choice



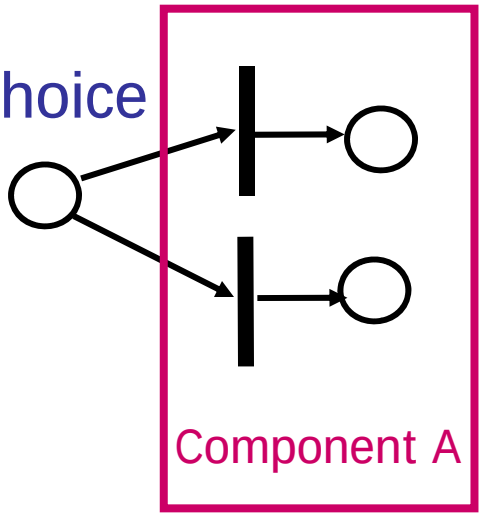
non-free choice

with conflict place

Non-local choice



local choice



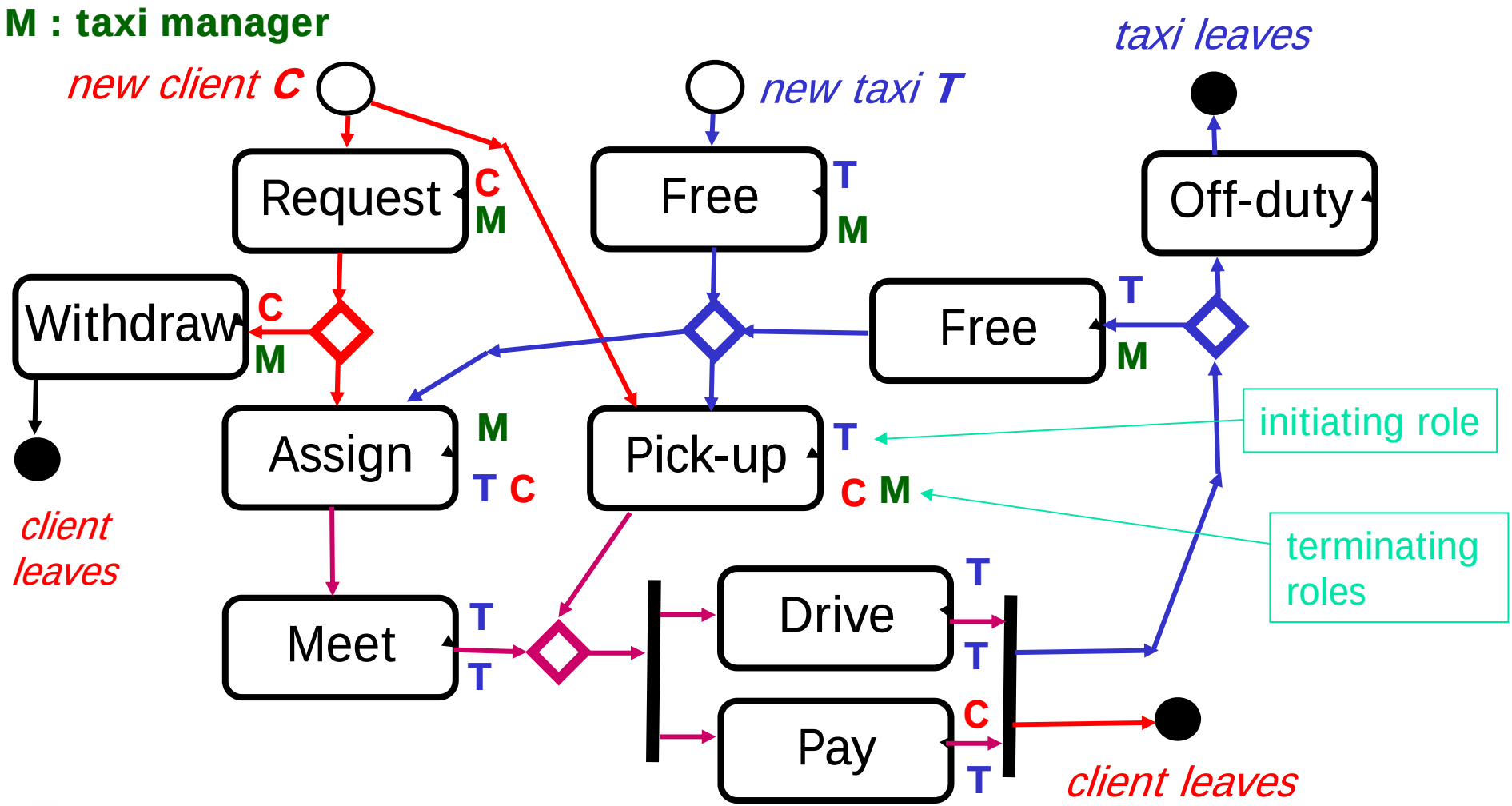
Sequence diagrams

- Sequence diagram (or Message Sequence Chart - MSC) is a well-known modeling paradigm showing a scenario of messages exchanged between a certain number of system components in some given order.
- **Limitation:** Normally, only a few of all the possible scenarios are shown.
- High-Level MSC can be used to describe the composition of MSCs (with weak sequencing – see below)

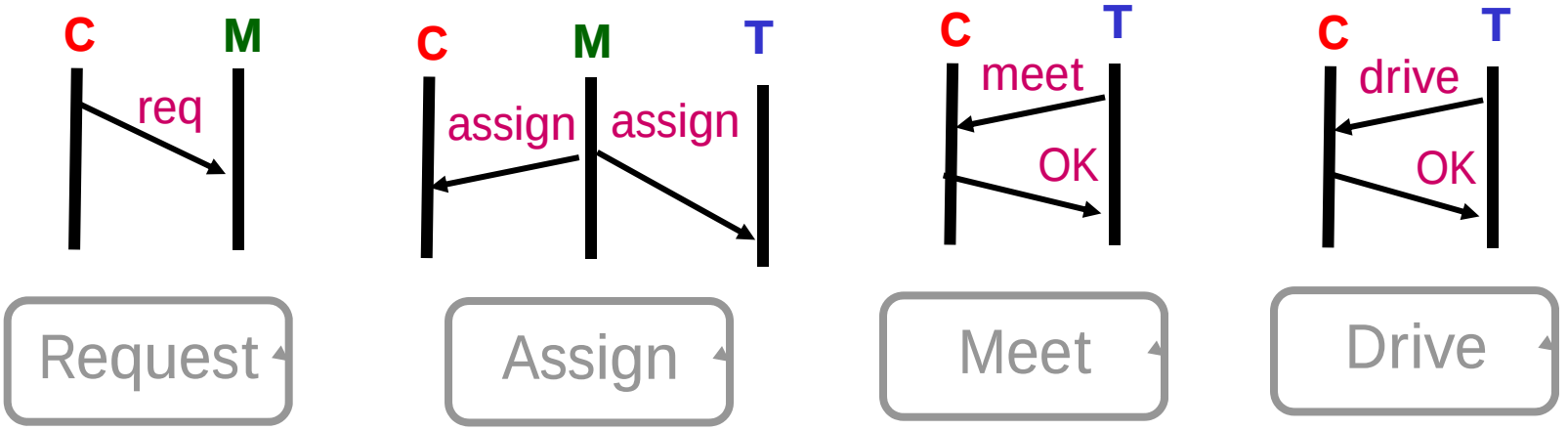
Example : a Taxi system

- Three roles (components):
  - User
  - Taxi
  - Manager
- Two use cases:
  - Normal: user requests a taxi from the manager, taxi assigned, meet, drive, pay
  - Street pick-up: user sees a taxi and the taxi stops and picks up the user, drive, pay
- Note: We assume that the three parties communicate through a specific application running on mobile devices.

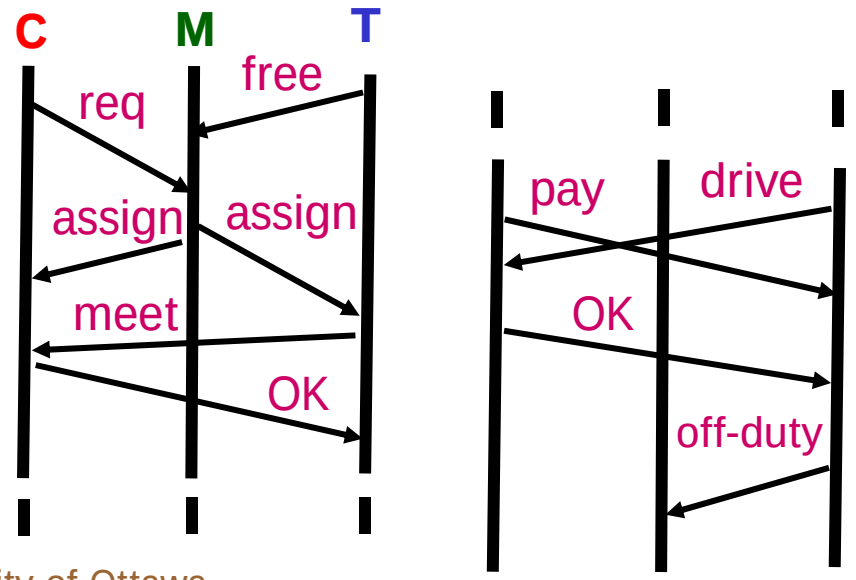
Example: Taxi system (an activity diagram - each activity is a collaboration between several roles: Client, Taxi, Manager)



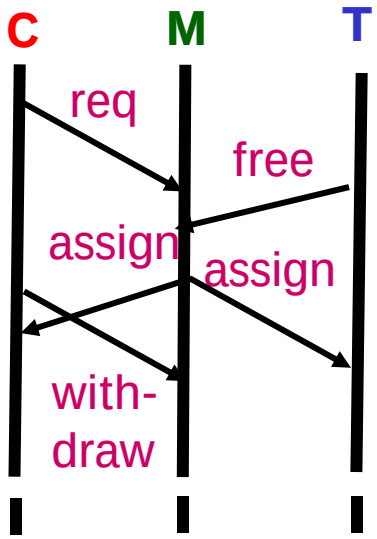
Taxi System
Detailed definitions of collaborations



Example scenario
(sequence diagram)

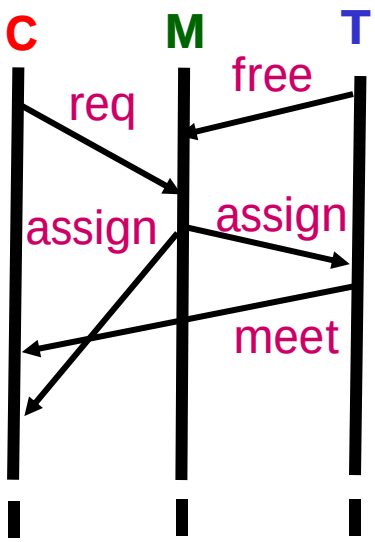


Taxi System : Problematic scenarios

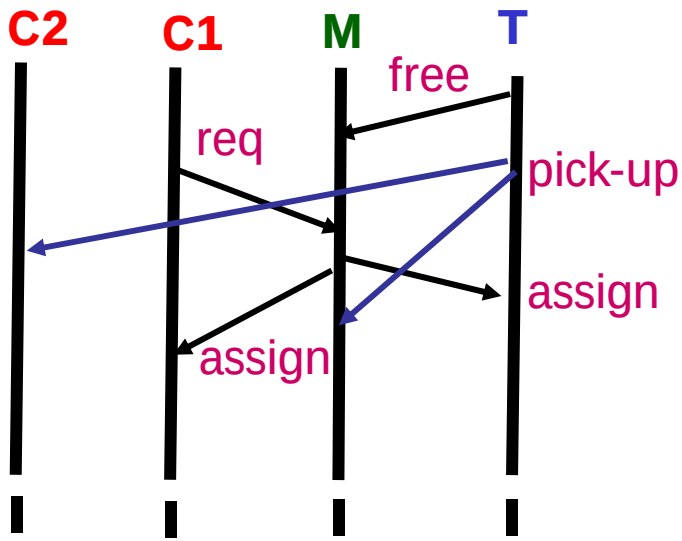


non-local
choice

[Gouda 84] suggests:
define different priorities
for different roles



race
condition



non-local
Choice
(conflict over taxi)

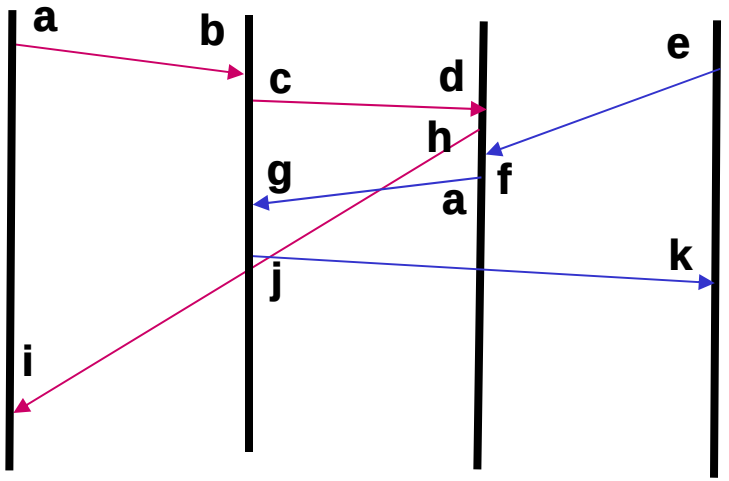
"implied scenario":

[Alur 2000] component behaviors
that realize the normal scenario
will also give rise to implied scenarios

Partial order of events

- Lamport [1978] pointed out that in a distributed system, there is in general no total order of events, only a partial order.
  - The events taking place at a given component can be totally ordered (assuming sequential execution).
  - The reception of a message is **after** its sending.
  - The after-relation is transitive.

For example, we have b after a and c after b; but d and e are unrelated (no order defined - concurrent), also j and i are concurrent. d after a by transitivity.



Strong and Weak sequencing

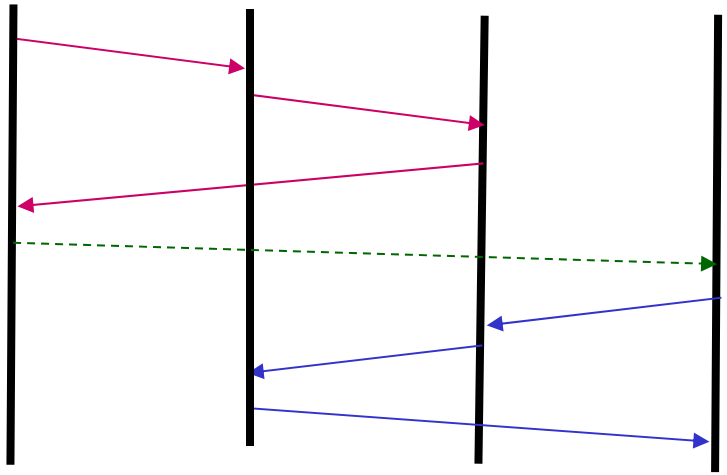
- Normal (strong) sequencing: C1 ; C2
  - all actions of C1 must be completed before any action of C2 may start.

Weak sequencing (introduced for the High-Level MSCs) is based on partial order.

- Weak sequencing: C1 $;_w$ C2
  - for each component c, all actions of C1 at c must be completed before any action of C2 at c may start.

(only local sequencing is enforced by each component, no global sequencing – this often leads to race conditions)

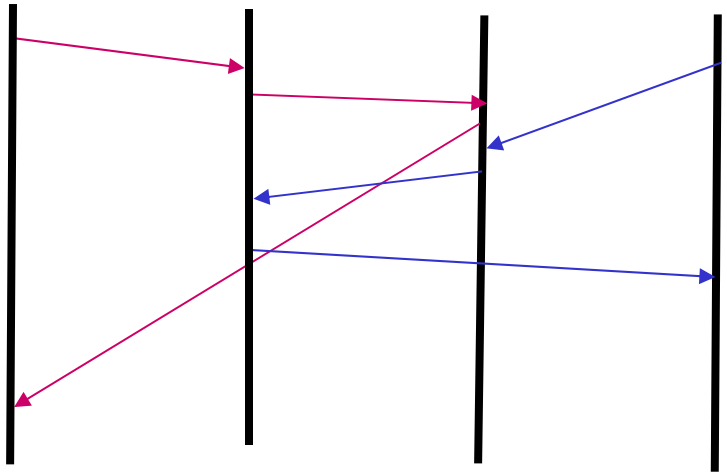
Example of strong and weak sequencing



strongly sequenced
(blue after red)

can be enforce by

→ **Coordination message**



weakly sequenced
(blue after$_w$ red)

how to enforce ?
(there are often race conditions)

3. Deriving component behaviors

Do you remember the problem ?

The problem

(early phase of the software development process)

- Define
  - Global functional requirements
  - Non-functional requirements
- Make high-level architectural choices
  - Identify system components
  - Define underlying communication service
- Define behavior of system components:
  - Locally performed functions
  - Communication protocol
    - Required messages to be exchanged and order of exchanges
    - Coding of message types and parameters

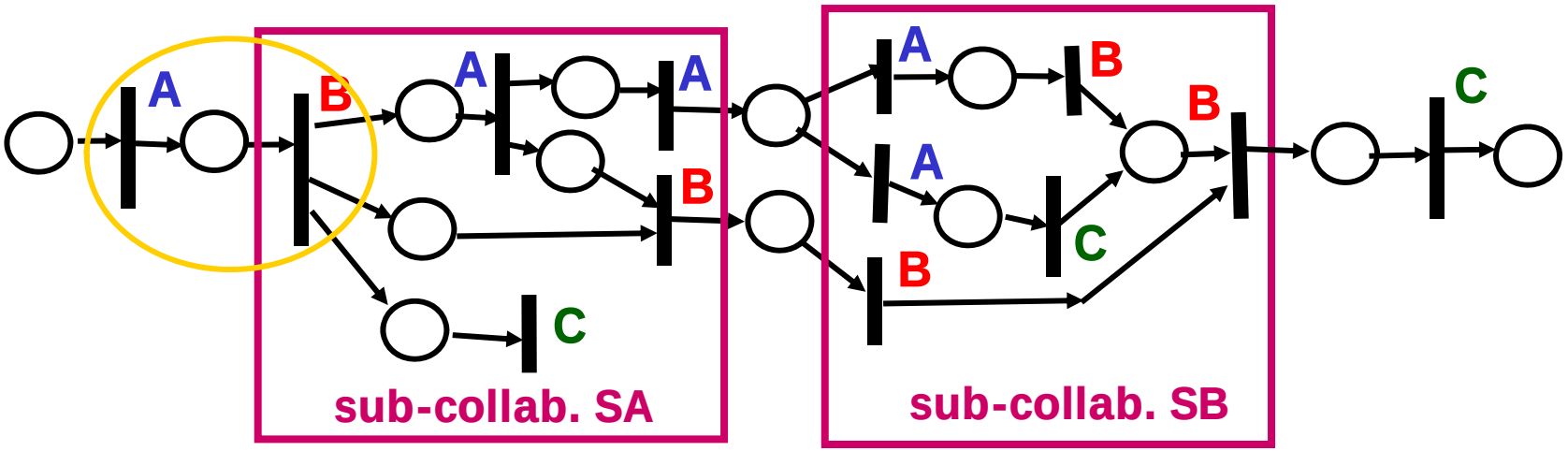
3.1 Distributed workflows

- We consider the following situation:
  - The global dynamic behavior is defined by an Activity diagram (or a similar notation) where each activity either represent a local action at a single component or a collaboration among several components.
  - Each explicit flow relations defines a partial order between a terminating actions of one activity and an initiating action of other activity.
  - Initially: No weak sequencing

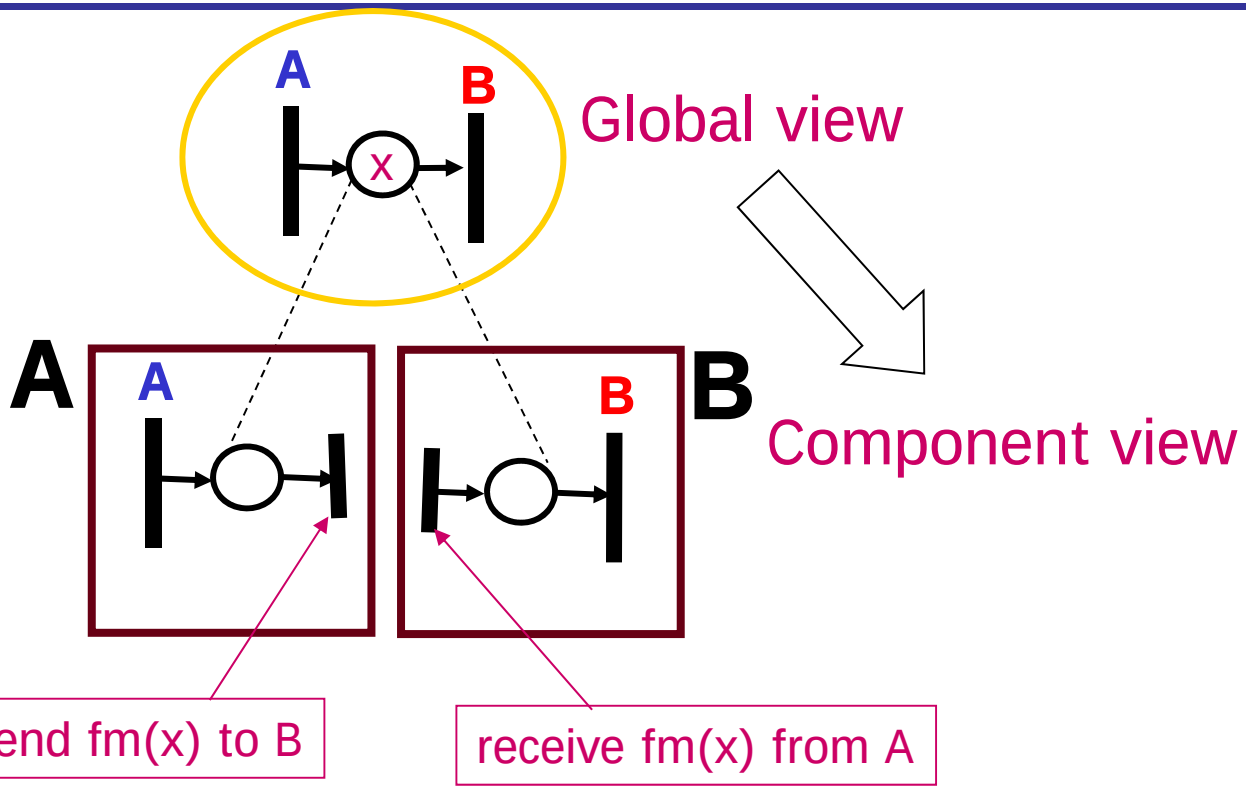
An example collaboration

Petri nets are a more simple formalism than Activity Diagrams. Therefore it is useful to first look for a general algorithm to derive component behaviors from global behavior specifications in the form of a Petri net.

There are three components: A, B and C

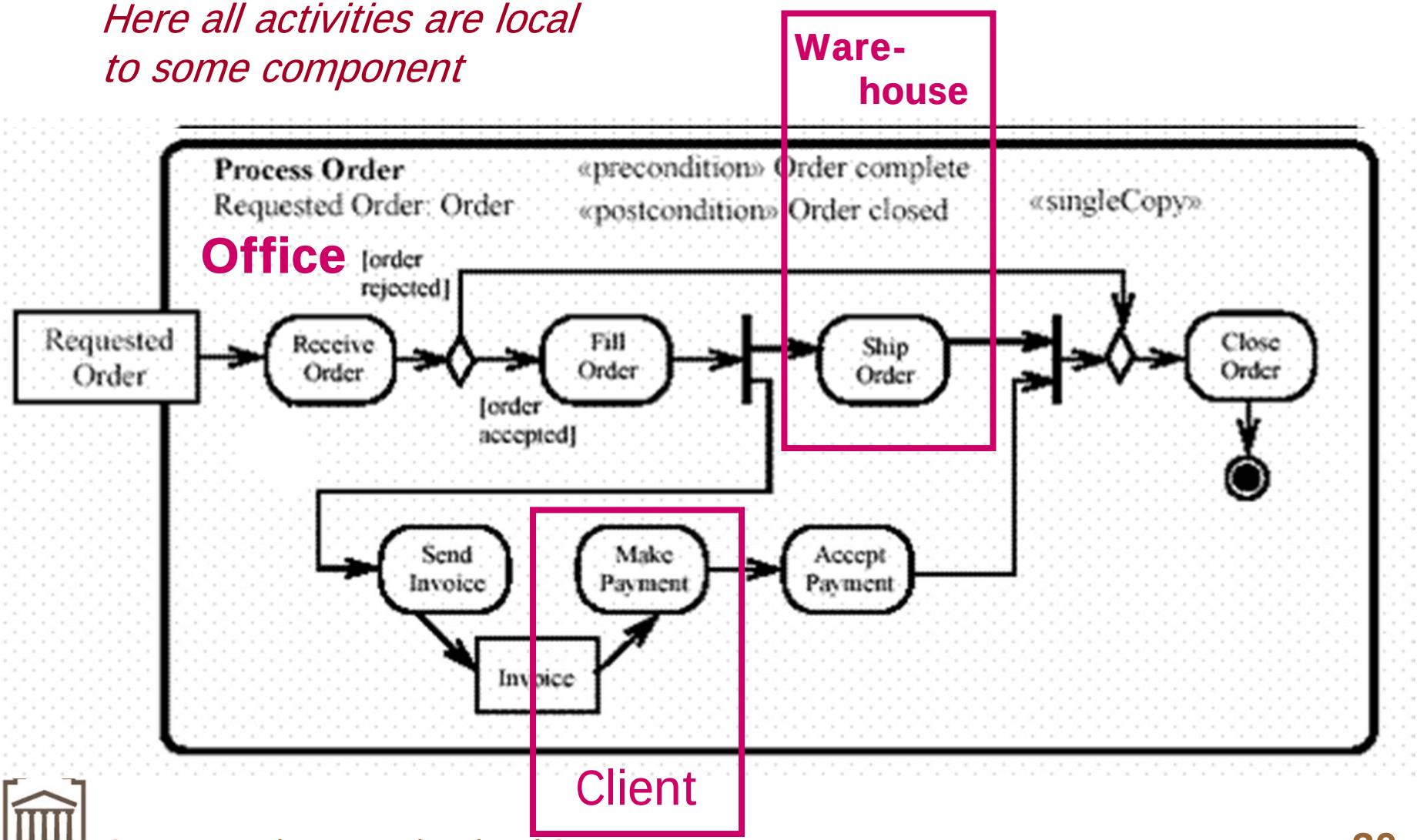


Component derivation rule



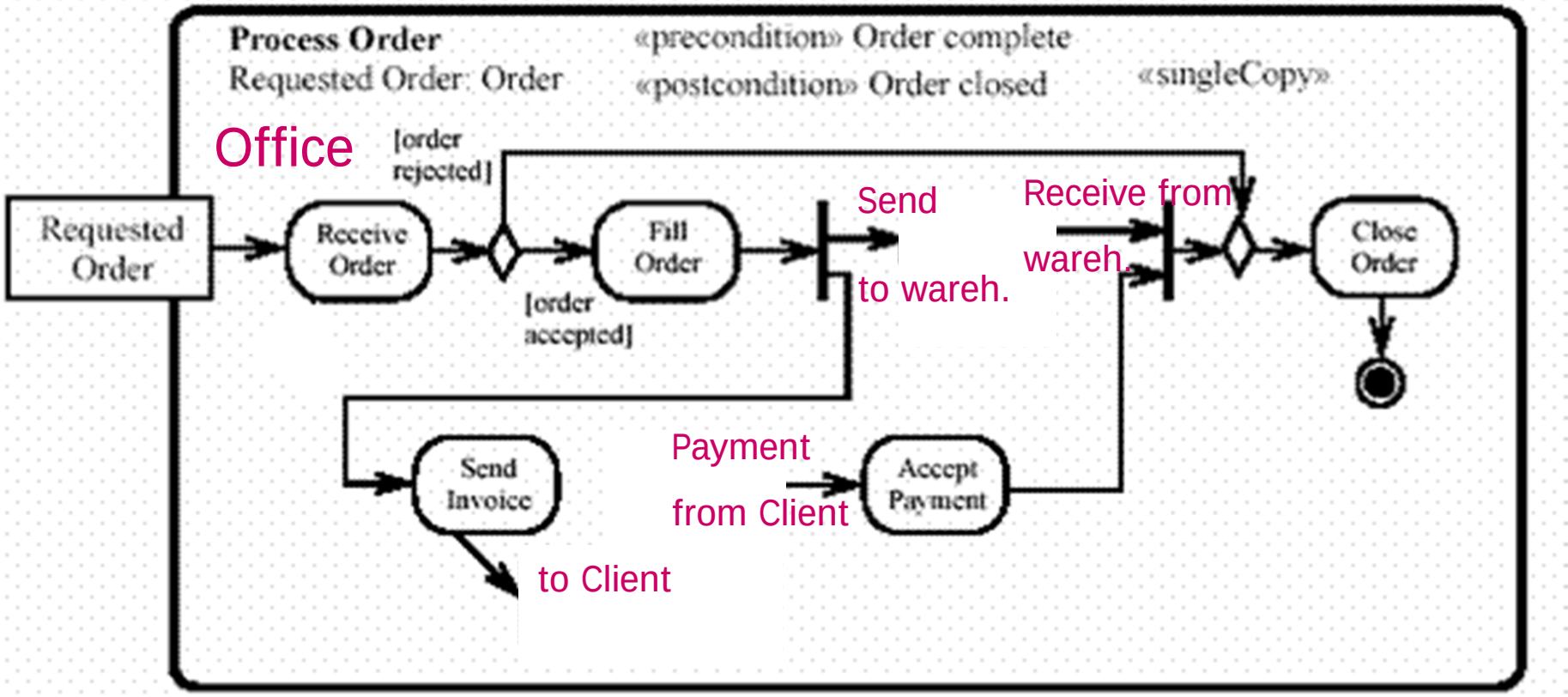
Example Activity Diagram

Here all activities are local to some component

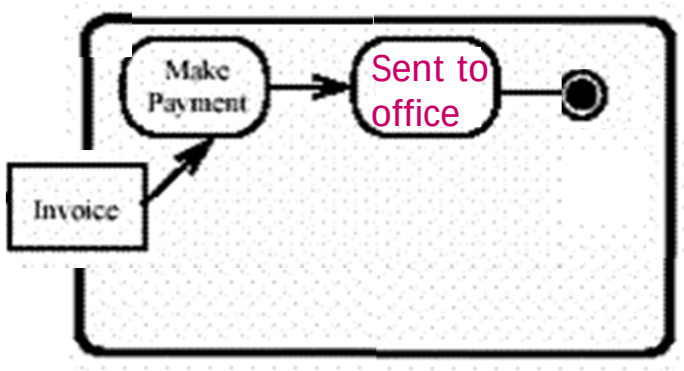


Office component

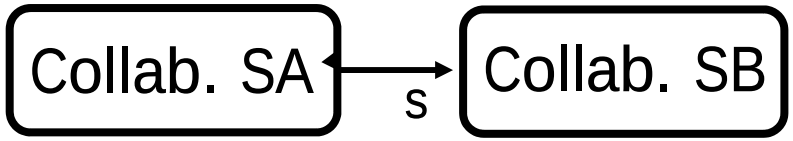
If a partial order relation goes from one component to another, then it should give rise to a send and receive operation in the respective components.



Client component

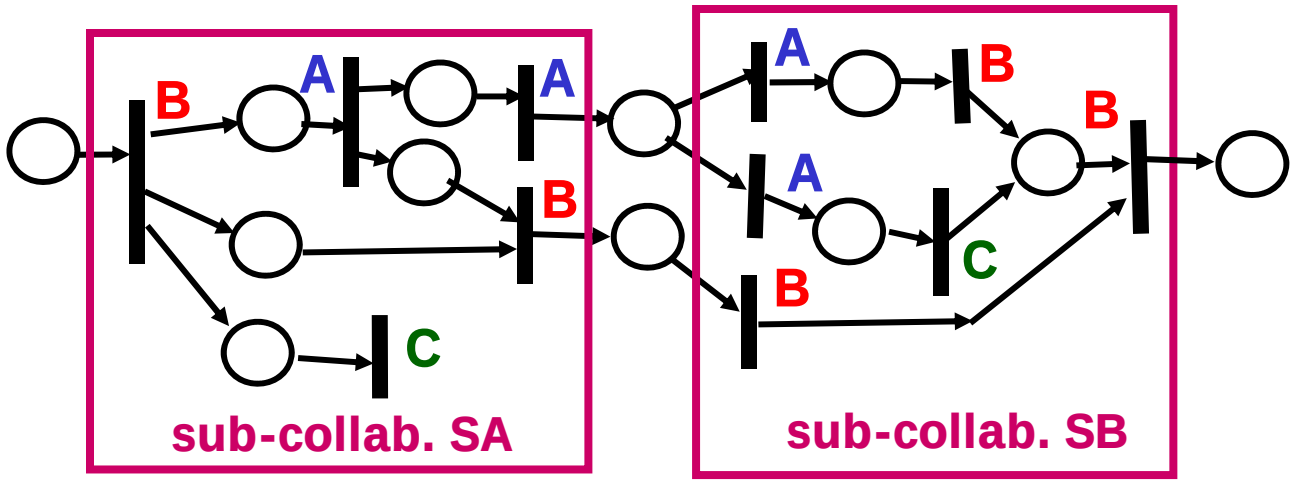


3.2. Strong sequencing between abstract sub-collaborations



This strong sequence means: *all actions of SA must be completed before actions of SB can start.*

The diagram below does not give strong sequencing: e.g. the transition of C of collaboration SA may occur after or during collaboration SB.



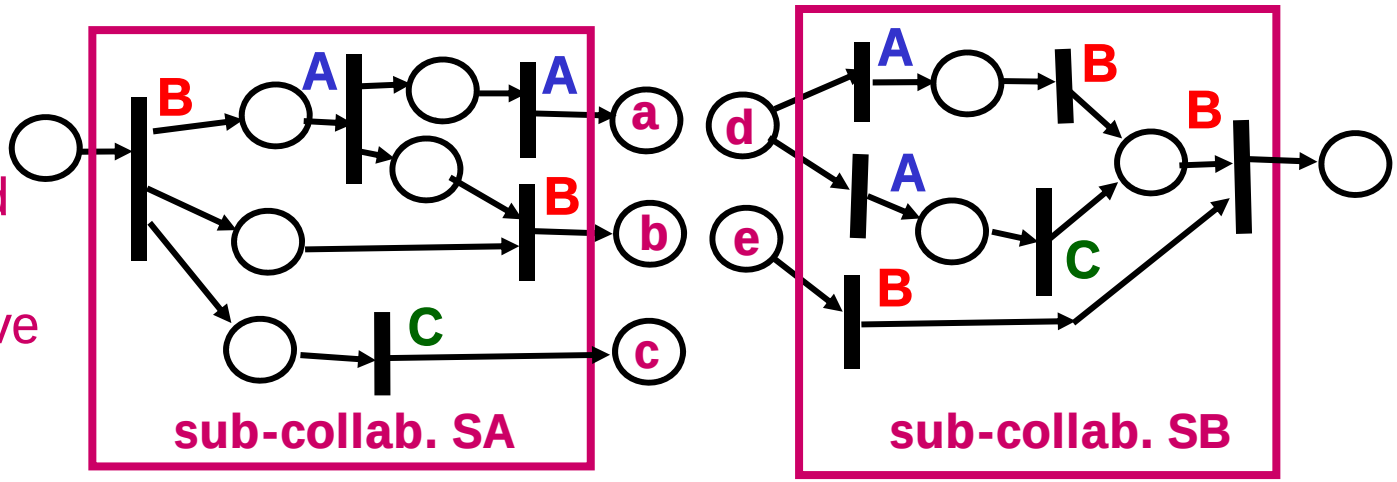
Initiating and terminating actions

- *initiating action* - no action is earlier (according to the partial order)
- *terminating actions* - no action is later

Strong sequencing SA ;$_s$ SB can be enforced by ensuring that all terminating actions of SA occur before all initiating actions of SB.

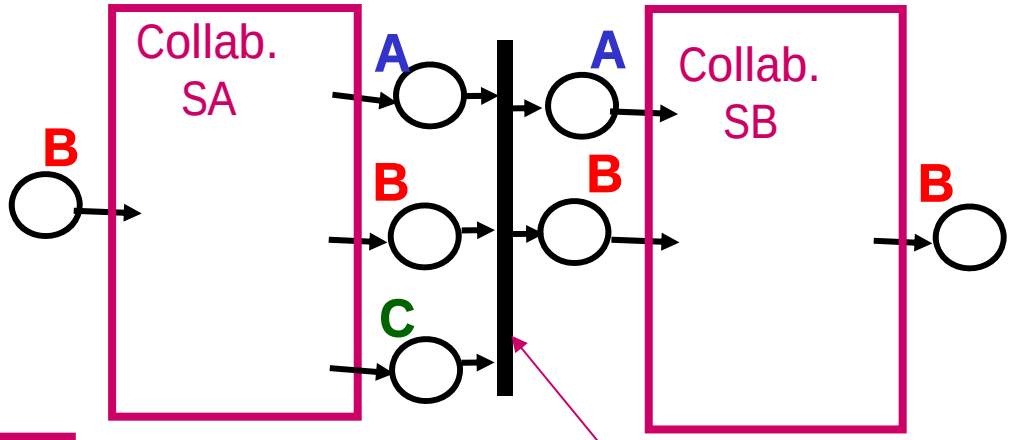
Transition C in SA is a terminating action.

Only after a, b and c have a token should tokens arrive in d and e.



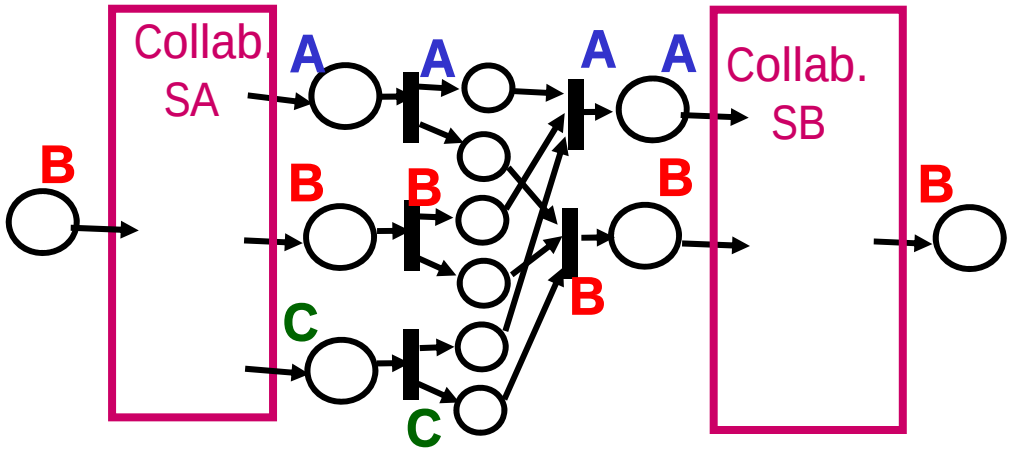
Realizing strong sequence

Two ways to coordinate the terminating and initiating actions:
centralized
and *distributed*



located at some given component

centralized realization

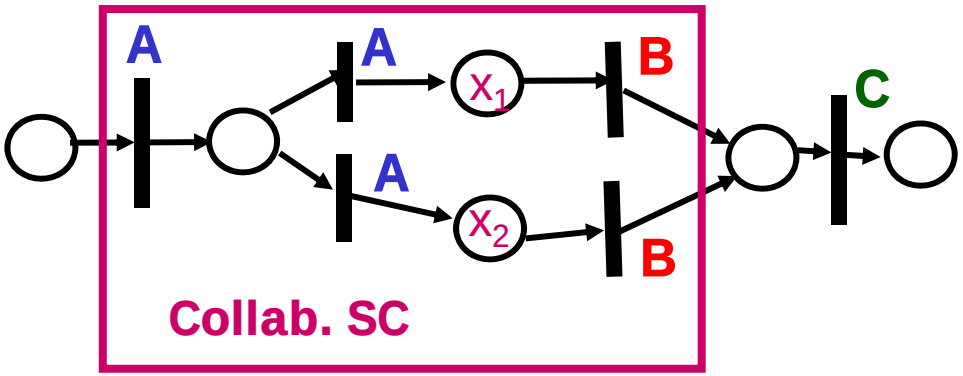


Distributed Realization
(first described in [Bochmann 86])

then apply derivation rule

Choice propagation

Here the choice is done by component A (local choice)



Component B should know which alternative was chosen
(include parameter x_i in flow message)

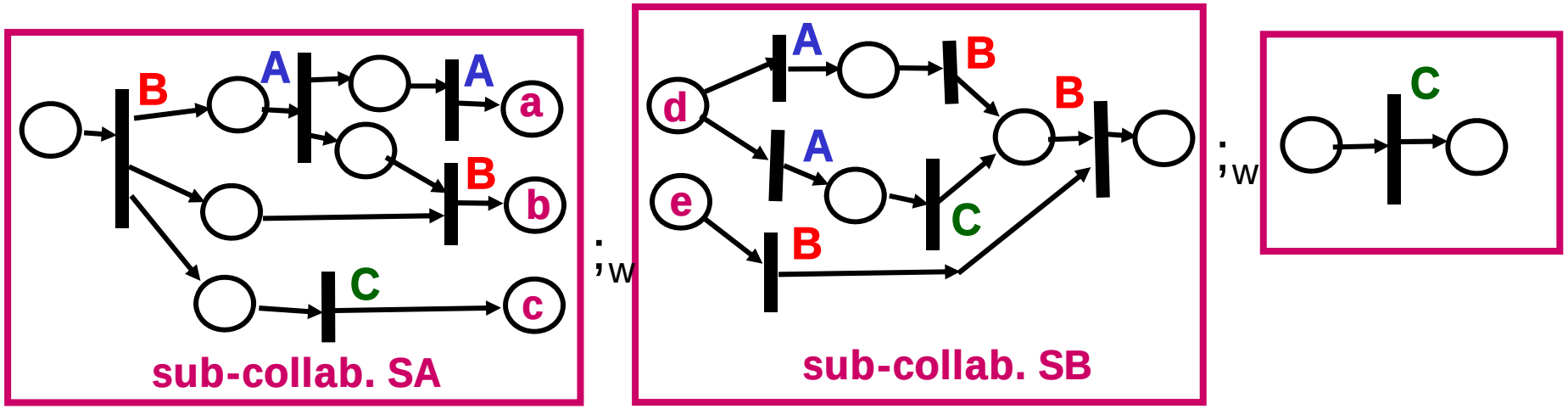
3.3. Component design for weak sequencing

- The component design approach described above was proposed in 1986 (see [Boch 86, Gotz 90])
- This was extended in 2008 to deal with weak sequencing [Bochmann 2008].

This new approach uses the ideas above and adds the following:

- Selective consumption of received messages
  - Received message enter a pool. The component fetches (or waits for) a given message when it is ready to consume it (like the Petri net models, see also [Mooij 2005])
- An additional type of message: choice indication message
- Additional message parameters, e.g. loop counters

Need for choice indication message (cim)



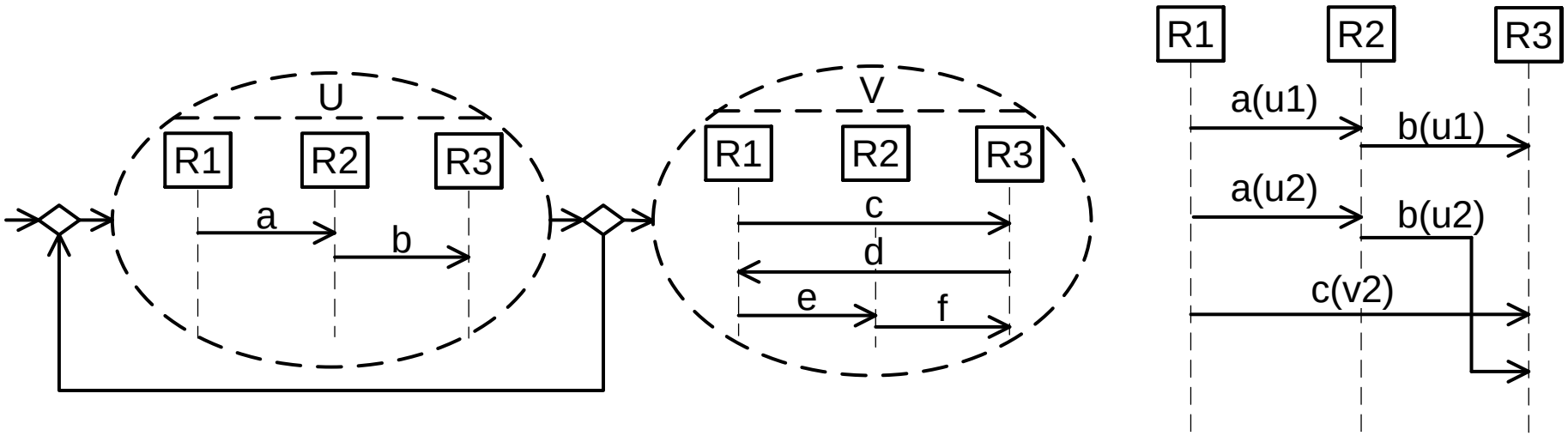
With weak sequencing, each component must know when the current sub-collaboration is locally complete in order to be ready to participate (or initiate) the next sub-collaboration.

• This is difficult for component C at the end of sub-collaboration B (above) if the upper branch was chosen (no message received).

Therefore we propose a **choice indication message**
(from A to C in this case)

Need for loop counters

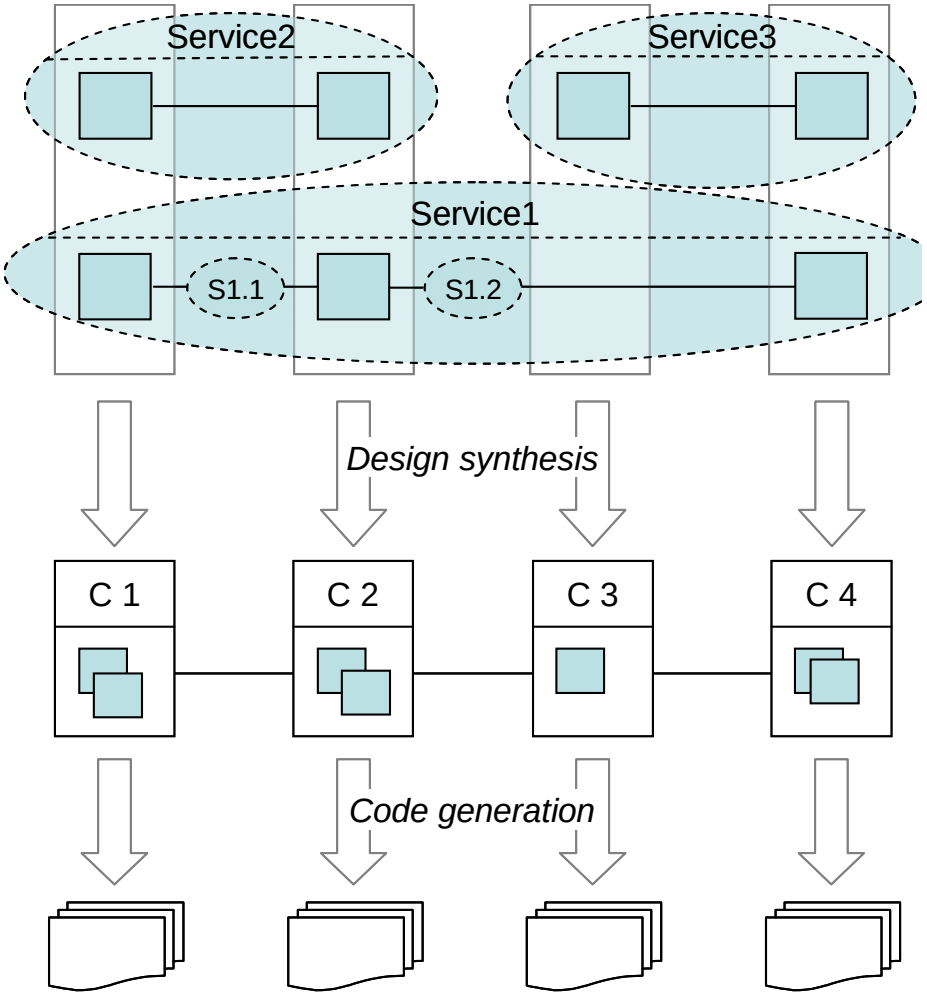
With weak sequencing, a message referring to the termination of a loop may arrive before a message referring to the last loop execution. See example:



Note: Nakata [1998] proposed to include in each coordination message an abbreviation of the complete execution history.

3.4 Summary

1. Define requirements in the form of a collaboration model
2. Architectural choices: allocate collaboration roles to different system components
3. Derive component behavior specifications (automated)
4. Evaluate performance and other non-functional requirements (revise architectural choices, if necessary)
5. Use automated tools to derive implementations of component behaviors.



Algorithm for deriving component behaviors

- Step 1: Calculate starting, terminating and participating roles for each sub-collaboration
- Step 2: Use architectural choices to determine starting, terminating and participating **components**.
- Step 3: For each component, use a recursively defined transformation function to derive the behavior of the component from the global requirements
  - Projection onto the given component
  - Additional flow and choice indication messages [Bochmann 2008]

Historical comments

- Initially, only strong sequencing, choice and concurrency operators, plus sub-behaviors
  [Bochmann and Gotzhein, 1986 and Gotzhein and Bochmann 1990]
  - Main conclusions:
    - Strong sequencing requires flow messages; need to identify initiating and terminating roles
    - Choice propagation: need for unique message parameters
- More powerful languages
  - LOTOS [Kant 1996]
    - recursive process call: >> ; disruption operator: [>
      (impossibility of distributed implementation)
  - Language with recursion and concurrency [Nakata 1998]

... for Petri nets

- Restriction: free-choice PN and “local choice” (as discussed above) [Kahlouche et al. 1996]
- general Petri nets [Yamaguchi et al. 2007]
  - It is quite complex (distributed choice of transition to be executed, depending on tokens in places associated with different sites)
- Note: These methods can be easily extended to Colored Petri nets (or Predicate Transition nets): exchanged messages now contain tokens with data parameters
- Petri nets with registers (see next slide) [Yamaguchi et al. 2003]

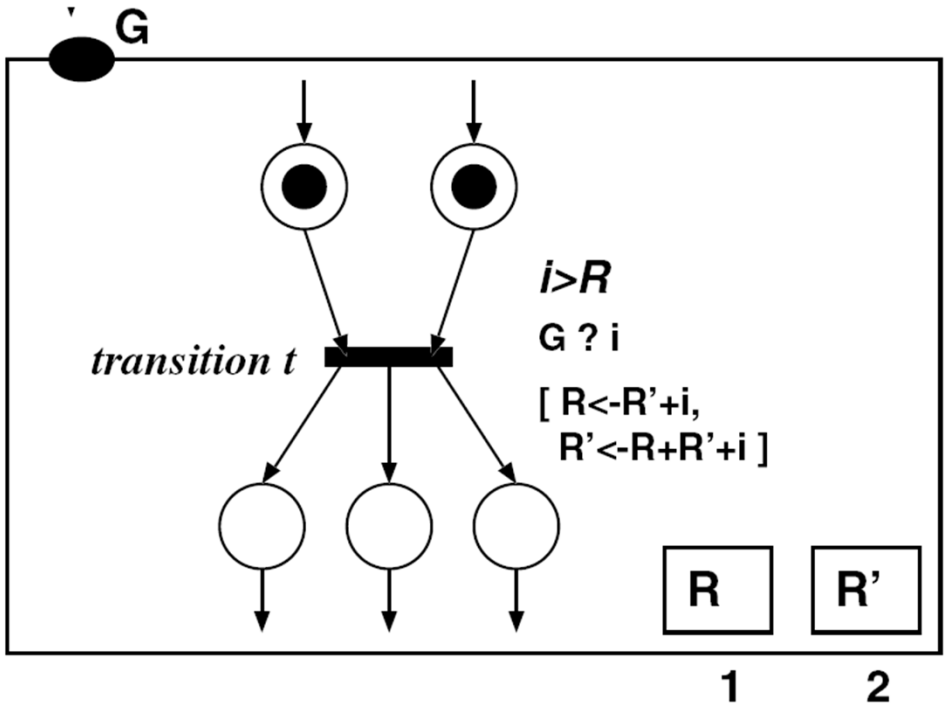
… for Petri net with registers

The Petri net has

- Local registers (e.g. R, R')

A transition has

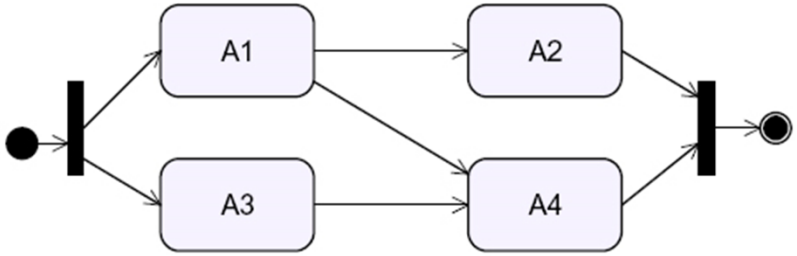
- External input or output interaction (e.g. G)
- Enabling predicate
- Update operations on registers



• The component behavior includes messages to exchange register values for evaluating predicates and updating registers.
• The number of required messages depends strongly on the distribution of the registers over the different components. – Optimization problem.

Remaining problems

- Support complex temporal order relationships with weak sequencing
  - Example:



- Data flow from non-terminating components
- Concurrent sessions and dynamic selection of collaboration partners
- Proof of correctness of derivation algorithm

Conclusions (i)

- **Distributed system design in several steps:**
  1. Requirements model: global behavior in terms of certain activities (collaborations) and their temporal ordering.
  2. Architectural choices: Based on architectural and non-functional requirements, allocate collaboration roles to system components
  3. Deriving component behavior (can be automated)
- **Proposed modeling language for requirements:**
  - Activity diagrams where an activity may be a collaboration between several roles
  - Identify roles for each activity (participating, starting, terminating)
  - Hierarchical description of requirements in terms of sub-activities (sub-collaborations)
- **Can be applied to other modeling languages:**
  - Hierachical State diagrams (UML)
  - Use Case Maps (standardized by ITU)
  - BPEL (business process execution language – for Web Services)
  - XPDL (Workflow Management Coalition) and BPMN (OMG)

Conclusions (ii)

- **Many fields of application:**
  - **service composition for communication services**
  - **workflows**
  - **e-commerce applications - Web Services**
  - **Grid and Cloud computing**
  - **Multi-core computer architectures**

- **Further work:**
  - **proving correctness of derivation algorithm**
  - **tools for deriving component behavior specifications**
  - **performance modeling for composed collaborations**
  - **“agile dynamic architectures”**

References

- [Alur 2000] Alur, Rajeev, Etessami, Kousha, & Yannakakis, Mihalis. 2000. Inference of message sequence charts. *Pages 304–313 of: 22nd International Conference on Software Engineering (ICSE'00)*.
- [Boch 86g] G. v. Bochmann and R. Gotzhein, Deriving protocol specifications from service specifications, Proc. ACM SIGCOMM Symposium, 1986, pp. 148-156.
- [Bochmann 2008] G. v. Bochmann, Deriving component designs from global requirements, Proc. Intern. Workshop on Model Based Architecting and Construction of Embedded Systems (ACES), Toulouse, Sept. 2008.
- [Castejon 2007] H. Castejón, R. Bræk, G.v. Bochmann, Realizability of Collaboration-based Service Specifications, Proceedings of the 14th Asia-Pacific Soft. Eng. Conf. (APSEC'07), IEEE Computer Society Press, pp. 73-80, 2007.
- [Castejon 2011] H. N. Castejòn, G. v. Bochmann and R. Braek, On the realizability of collaborative services, Journal of Software and Systems Modeling, Vol. 10 (12 October 2011), pp. 1-21.
- [Gotz 90a] R. Gotzhein and G. v. Bochmann, Deriving protocol specifications from service specifications including parameters, ACM Transactions on Computer Systems, Vol.8, No.4, 1990, pp.255-283.
- [Goud 84] M. G. Gouda and Y.-T. Yu, Synthesis of communicating Finite State Machines with guaranteed progress, IEEE Trans on Communications, vol. Com-32, No. 7, July 1984, pp. 779-788.
- [Lamport 1978] L. Lamport, "Time, clocks and the ordering of events in a distributed system", *Comm. ACM*, 21, 7, July, 1978, pp. 558-565.
- [Kant 96a] C. Kant, T. Higashino and G. v. Bochmann, Deriving protocol specifications from service specifications written in LOTOS, Distributed Computing, Vol. 10, No. 1, 1996, pp.29-47.
- [Mouij 2005] A. J. Mooij, N. Goga and J. Romijn, "Non-local choice and beyond: Intricacies of MSC choice nodes", *Proc. Intl. Conf. on Fundamental Approaches to Soft. Eng. (FASE'05)*, LNCS, 3442, Springer, 2005.
- [Nakata 98] A. Nakata, T. Higashino and K. Taniguchi, "Protocol synthesis from context-free processes using event structures", *Proc. 5th Intl. Conf. on Real-Time Computing Systems and Applications (RTCSA'98)*, Hiroshima, Japan, IEEE Comp. Soc. Press, 1998, pp.173-180.
- [Sanders 05] R. T. Sanders, R. Bræk, G. v. Bochmann and D. Amyot, "Service discovery and component reuse with semantic interfaces", *Proc. 12th Intl. SDL Forum*, Grimstad, Norway, LNCS, vol. 3530, Springer, 2005.
- [Yama 03a] H. Yamaguchi, K. El-Fakih, G. v. Bochmann and T. Higashino, Protocol synthesis and re-synthesis with optimal allocation of resources based on extended Petri nets., Distributed Computing, Vol. 16, 1 (March 2003), pp. 21-36.
- [Yama 07] H. Yamaguchi, K. El-Fakih, G. v. Bochmann and T. Higashino, Deriving protocol specifications from service specifications written as Predicate/Transition-Nets, Computer Networks, 2007, vol. 51, no1, pp. 258-284

Thanks !

Any questions ??

For copy of slides, see

http://www.site.uottawa.ca/~bochmann/talks/Deriving.ppt