# Modelling and Verification of Component Compatibility by Composition

MOCA'04
Third Workshop on Modelling of Objects, Components and Agents

Aarhus, Denmark
October 11-13, 2004

Donald C. Craig
and
Wlodek M. Zuberek

Department of Computer Science
Memorial University of Newfoundland
St. John's, Canada A1B 3X5

# Complexity in Software Engineering

- Construction of large-scale software projects is becoming increasingly difficult as architectures become more sophisticated.

- To combat complexity, there has been a natural, evolutionary trend from low-level constructs to higher-level abstractions in the software engineering process:

  - structured programming
  - object-oriented programming
  - aspect-oriented programming
  - architecture description languages

- In recent years, *component-based* software engineering has been proposed as a means of mitigating the complexities associated with construction of large software architectures.

# Component Background

- Informal definitions of components are numerous. For example:
  - "An independently deliverable piece of functionality providing access to its services through interfaces."

    Alan W. Brown (2001) *An Overview of Components and Component-Based Development*.

  - "A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties."

    Clemens Szyperski (2002) *Component Software : Beyond Object-Oriented Programming (second edition)*.

- Before an attempt can be made to automate analysis of component systems, formal definitions of *component* must be proposed. Such definitions are beginning to appear in the literature.

- *Component compatibility* is required for reusability and substitutibility in software development and maintenance.

# Petri Net Interface Model

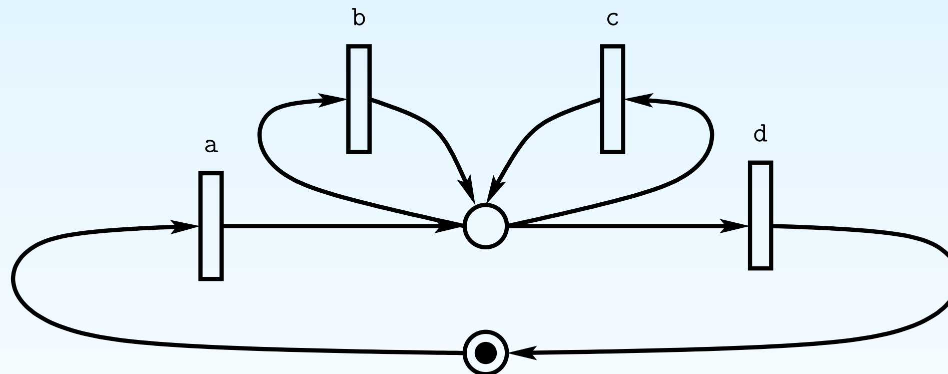
- It is assumed that internal behavioural semantics of components are unimportant and the behaviour can be described as a set of service sequences (or a set of sequences of interface operations).

- This behaviour is represented by a (cyclic) labelled Petri net:

$$\mathcal{M}_i = (P_i, T_i, A_i, S_i, \ell_i, m_i), \quad \ell_i : T_i \to S_i, \quad m_i : P_i \to \{ 0, 1, \ldots \}.$$

  Different services are represented by different labels from the set $S_i$.

- Component interactions occur between requester interfaces (*r-interfaces*) and provider interfaces (*p-interfaces*). The same component can have several r-interfaces and several p-interfaces.



# Petri Net Model — Interface Languages

- The behaviour of an interface $\mathcal{M}_i$ is described by a protocol or language which is the set of all (initial) firing sequences, $\mathcal{F}(\mathcal{M}_i)$.

- A single firing sequence $\sigma$ from this set is:

$$\sigma = t_{i_1} t_{i_2} \ldots\ t_{i_k} \Leftrightarrow (\forall\ 0 < j \leq k : t_{i_j} \in E(m_{i_{j-1}}) \wedge m_{i_{j-1}} \stackrel{t_{i_j}}{\rightarrow} m_{i_j}) \wedge m_{i_0} = m_i,$$

where $E(m)$ is the set of transitions enabled by the marking $m$.

- The language of $\mathcal{M}_i$, denoted by $\mathcal{L}(\mathcal{M}_i)$, is defined as:

$$\mathcal{L}(\mathcal{M}_i) = \{\ \ell(\sigma) \mid \sigma \in \mathcal{F}(\mathcal{M}_i) \wedge \ell(\sigma) \text{ is a complete sequence of operations } \},$$

where $\ell(t_{i_1} \ldots t_{i_k}) = \ell(t_{i_1}) \ldots \ell(t_{i_k})$ and a *complete sequence of operations* represents a session of requester/provider interactions (*i.e.*, a cycle of an interface model).

# Interface Composition

An r-interface $\mathcal{M}_i = (P_i, T_i, A_i, S_i, \ell_i, m_i)$ composed with a p-interface $\mathcal{M}_j = (P_j, T_j, A_j, S_j, \ell_j, m_j)$ creates a new net $\mathcal{M}_{ij} = (P_{ij}, T_{ij}, A_{ij}, S_i, \ell_{ij}, m_{ij})$, provided that $S_i \subseteq S_j$ and where:

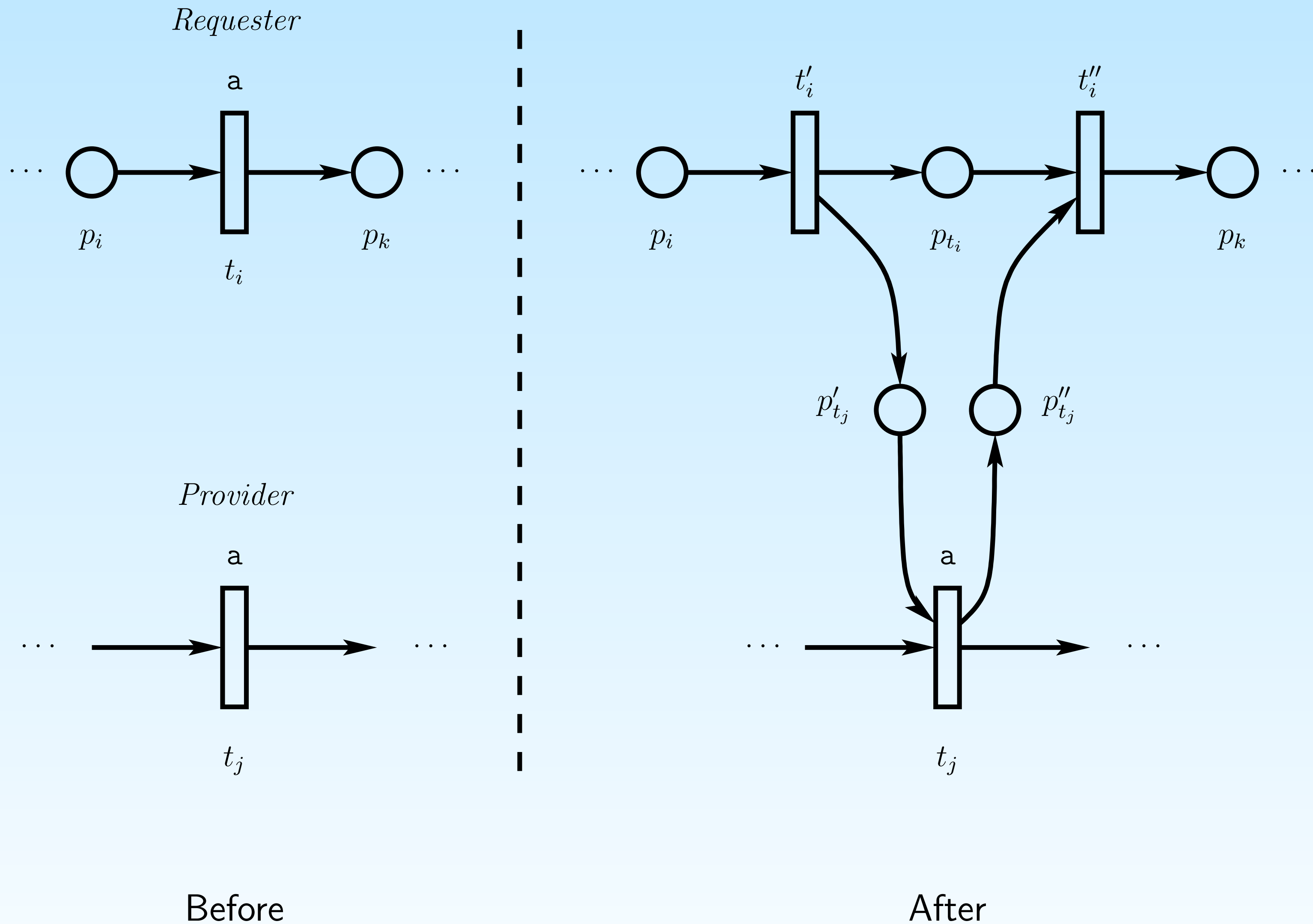
$$
\begin{aligned}
P_{ij} &= P_i \cup P_j \cup \{\, p_{t_i} : t_i \in \hat{T}_i \,\} \cup \{\, p'_{t_j}, p''_{t_j} : t_j \in \hat{T}_j \,\}, \\
&\quad \text{where } \hat{T}_i = \{\, t \in T_i : \ell_i(t) \neq \varepsilon \,\},\ \hat{T}_j = \{\, t \in T_j : \ell_j(t) \neq \varepsilon \,\}, \\
T_{ij} &= T_i \cup T_j - \hat{T}_i \cup \{\, t'_i, t''_i : t_i \in \hat{T}_i \,\}, \\
A_{ij} &= A_i \cup A_j - P_i \times \hat{T}_i - \hat{T}_i \times P_i \cup \\
&\quad \{\, (p_i, t'_i), (t'_i, p_{t_i}), (p_{t_i}, t''_i), (t''_i, p_k), (t'_i, p'_{t_j}), (p'_{t_j}, t_j), (t_j, p''_{t_j}), (p''_{t_j}, t''_i) : \\
&\qquad t_i \in \hat{T}_i \ \wedge\ t_j \in \hat{T}_j \ \wedge\ \ell_i(t_i) = \ell_j(t_j) \ \wedge\ (p_i, t_i) \in A_i \ \wedge\ (t_i, p_k) \in A_i \,\},
\end{aligned}
$$

$$
\forall t \in T_{ij} : \ell_{ij}(t) = \begin{cases} \ell_i(t), & \text{if } t \in T_i, \\ \ell_j(t), & \text{if } t \in T_j, \\ \varepsilon, & \text{otherwise;} \end{cases}
$$

$$
\forall p \in P_{ij} : m_{ij}(p) = \begin{cases} m_i(p), & \text{if } p \in P_i, \\ m_j(p), & \text{if } p \in P_j, \\ 0, & \text{otherwise.} \end{cases}
$$

# Interface Composition (cont'd)

A composition, for a single interface operation, can be illustrated as follows:



# Component Compatibility

- In the composed net, all (nontrivially) labelled transitions are shared by both interfaces.

- Any string generated by the resulting composition can also be generated by each interface:

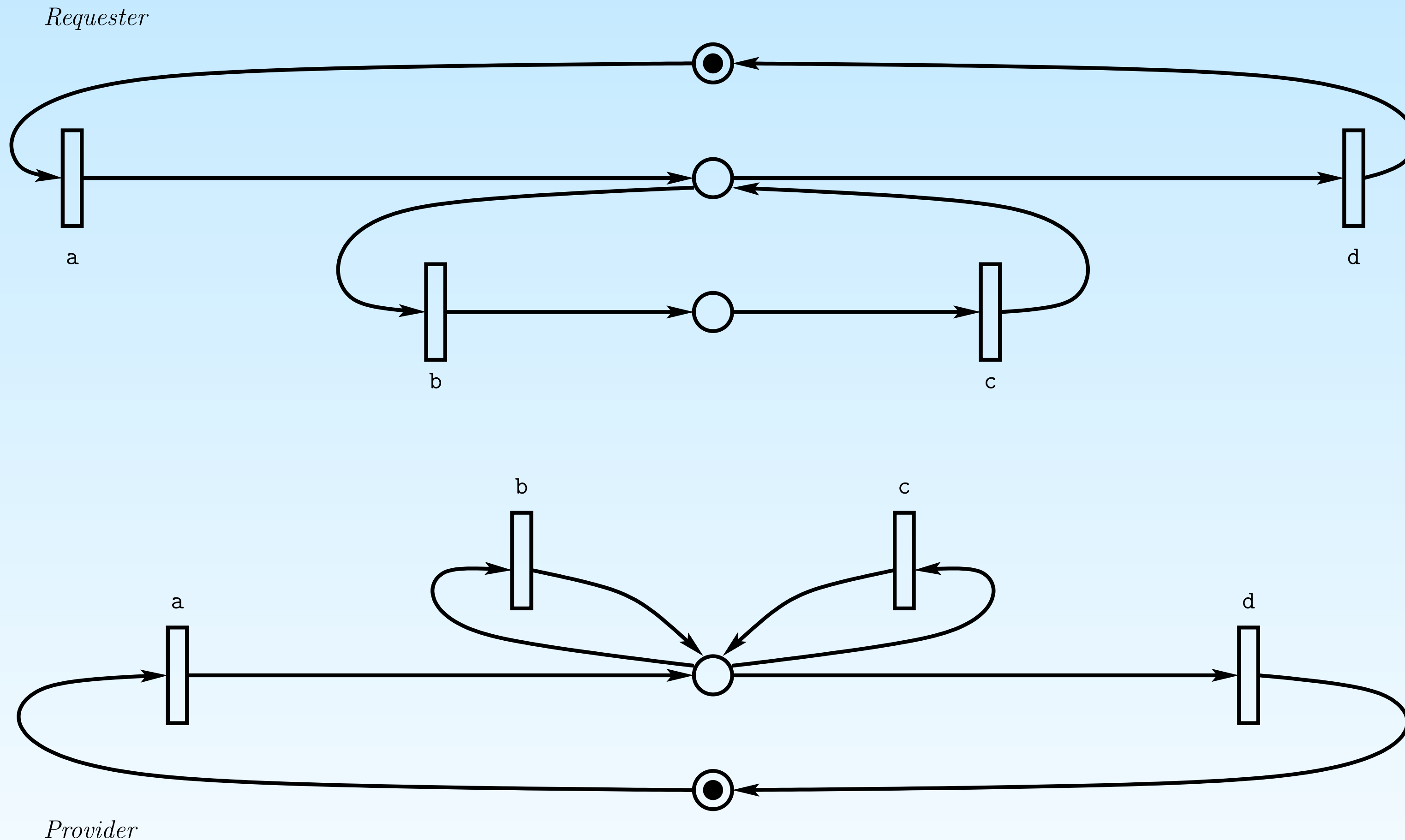
  The language of the composition of two interfaces with the same alphabet $S$, an r-interface $\mathcal{M}_i$ and a p-interface $\mathcal{M}_j$, $\mathcal{M}_i \rhd \mathcal{M}_j$, is the intersection of $\mathcal{L}(\mathcal{M}_i)$ and $\mathcal{L}(\mathcal{M}_j)$, $\mathcal{L}(\mathcal{M}_i \rhd \mathcal{M}_j) = \mathcal{L}(\mathcal{M}_i) \cap \mathcal{L}(\mathcal{M}_j)$.

- The compatibility of two components can be checked by detecting deadlocks in the composed net:

  Two interfaces with the same alphabet $S$, an r-interface $\mathcal{M}_i$ and a p-interface $\mathcal{M}_j$ are incompatible iff the composition $\mathcal{M}_{ij}$ = $\mathcal{M}_i \rhd \mathcal{M}_j$ contains a deadlock.

# Example 1

Consider database client and server components. The server (provider) supports an *open* operation $(\mathtt{a})$, followed by any number of *read/write* operations in any order $(\mathtt{b}|\mathtt{c})^*$ followed by a *close* operation $(\mathtt{d})$. The interface language of the client (requester) is a subset of this language.



# Example 1 (cont'd)

Provider language $\mathcal{L}_P = (\mathtt{a}(\mathtt{b}|\mathtt{c})^*\mathtt{d})^*$; requester language $\mathcal{L}_R = (\mathtt{a}(\mathtt{bc})^*\mathtt{d})^*$

Note: $\mathcal{L}_R \subseteq \mathcal{L}_P$, so $\mathcal{L}_R \cap \mathcal{L}_P = \mathcal{L}_R$.

$\mathcal{M}_{ij}$ = $\mathcal{M}_i \triangleright \mathcal{M}_j$ is deadlock-free $\Rightarrow$ $\mathcal{M}_i$ is compatible with $\mathcal{M}_j$.

*Requester*

b

c

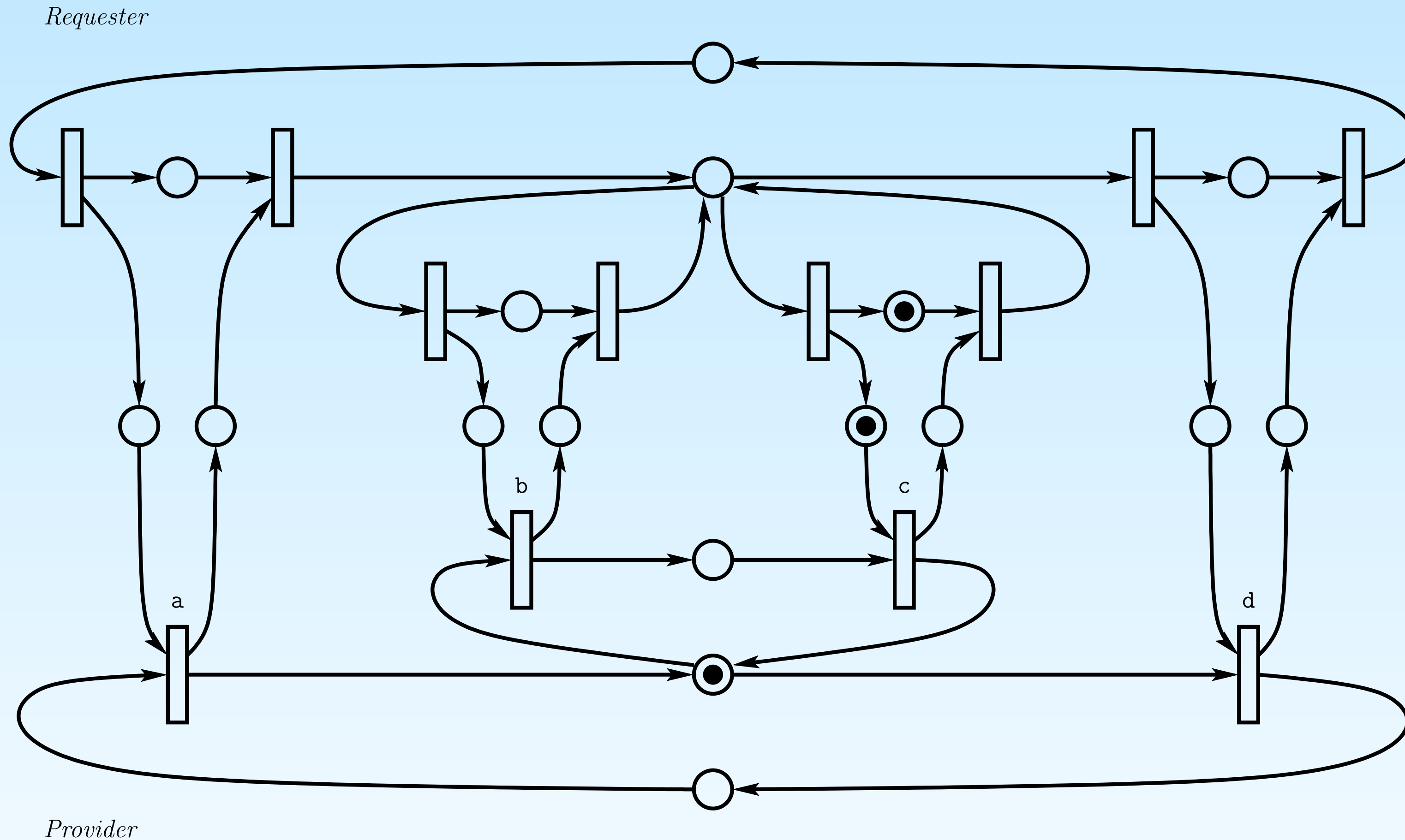
a

d

*Provider*

# Example 1 (cont'd)

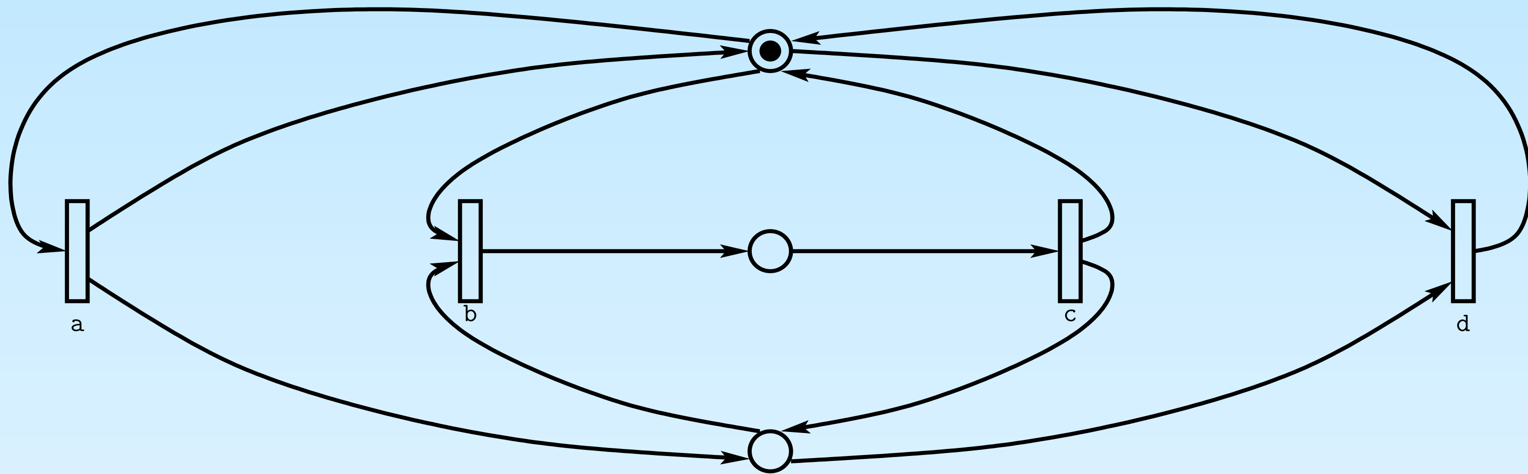
Swapping the provider and requester in the previous example, so $\mathcal{L}_P = (\mathtt{a}(\mathtt{bc})^*\mathtt{d})^*$ and $\mathcal{L}_R = (\mathtt{a}(\mathtt{b}|\mathtt{c})^*\mathtt{d})^*$, results in composition that exhibits a deadlock as shown below:

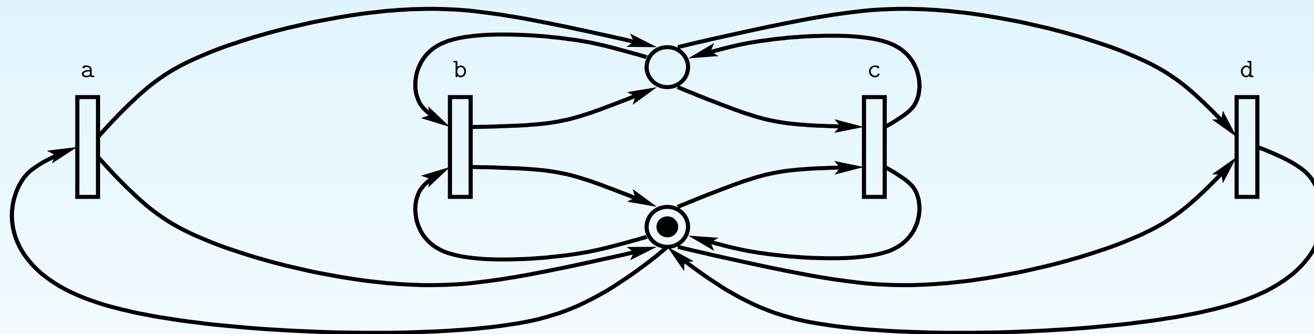



$\mathcal{L}_R \not\subseteq \mathcal{L}_P$, $\mathcal{M}_i$ is incompatible with $\mathcal{M}_j$.

# Example 2

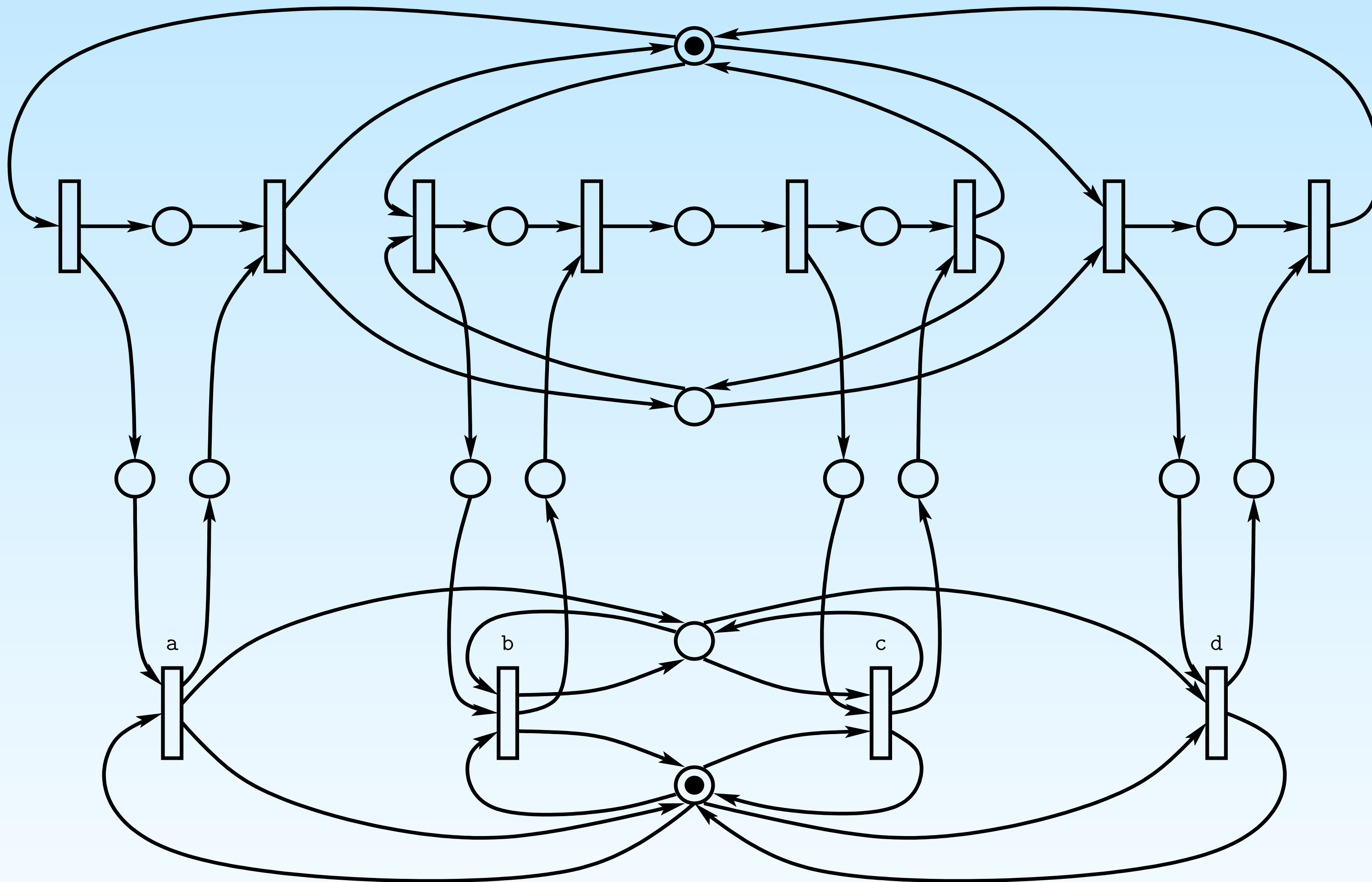
Context free languages (*e.g.* nested transactions on a database).
Before composition:



# Example 2 (cont'd)

Context free languages (*e.g.* nested transactions on a database).
After composition:



# Conclusions and Future work

- Static and dynamic component aspects must be understood in order to determine compatibility.

- Compatibility can be checked by representing the interface behaviours as Petri nets and then composing them.
  - If the resulting net exhibits deadlock, then the components are not compatible.
  - Deadlock detection — structural techniques can be applied given the presence of cyclic subnets.

- Hierarchical representations of component interfaces in complex software architectures follows the same approach.

- Dynamic reconfiguration/collaboration between components may be possible. (Self assembling software?)