

Component Compatibility and its Verification

Donald Craig Wlodek Zuberek

Department of Computer Science
Memorial University of Newfoundland
Canada

The First International Workshop on
Software Architecture Research and Practice
July 2, 2007 – Silicon Valley, USA

Introduction

- Construction of large-scale software projects is becoming increasingly difficult as architectures and requirements become more sophisticated.
- To combat complexity, there has been a trend from low-level constructs to higher-level abstractions in the software engineering process:
 - structured programming
 - object-oriented programming
 - agile software development
 - aspect-oriented programming
 - ...
- *Component-based software engineering* (CBSE) is becoming more popular as a means of mitigating the complexities associated with construction of large software architectures.
- One of the challenges of CBSE is the issue of integrating a collection of components together to form a functioning system.

Components

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

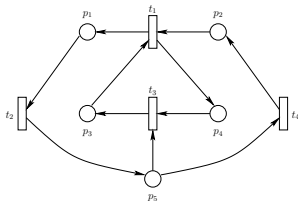
Alan W. Brown (2001) *An Overview of Components and Component-Based Development*.

- “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)*.

- Almost all informal definitions mention the concept of an *interface*, through which components interact with the external world.
- Before an attempt can be made to verify component compatibility, formal definitions of *component* must be proposed.
- *Petri nets* can be used to model components — in particular, their interface behaviours.

Introduction to Petri Nets



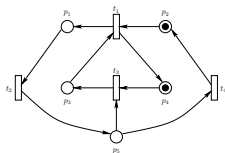
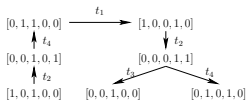
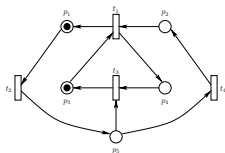
$$C = \begin{bmatrix} +1 & -1 & 0 & 0 \\ -1 & 0 & 0 & +1 \\ -1 & 0 & +1 & 0 \\ +1 & 0 & -1 & 0 \\ 0 & +1 & -1 & -1 \end{bmatrix}$$

- Structurally, a Petri net is defined by sets of places and transitions which are connected to each other by directed arcs, $\mathcal{N} = (P, T, A)$. We define:

$$\begin{aligned} \text{Inp}(p) &= \{ t \in T \mid (t, p) \in A \}, & e.g. \text{Inp}(p_5) &= \{t_2\} \\ \text{Out}(p) &= \{ t \in T \mid (p, t) \in A \}, & e.g. \text{Out}(p_5) &= \{t_3, t_4\} \\ \text{Inp}(t) &= \{ p \in P \mid (p, t) \in A \}, & e.g. \text{Inp}(t_1) &= \{p_2, p_3\} \\ \text{Out}(t) &= \{ p \in P \mid (t, p) \in A \}, & e.g. \text{Out}(t_1) &= \{p_1\}. \end{aligned}$$

- The structure of a Petri net can be described by a connectivity matrix.
- A siphon is a subset of places, S , such that $\text{Inp}(S) \subseteq \text{Out}(S)$. *e.g.* $\{p_3, p_4\}$ and $\{p_1, p_3, p_4\}$ are siphons (there are others).

Petri Nets



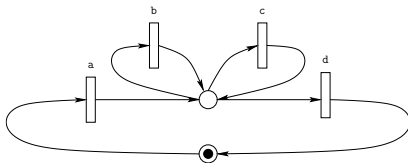
- A marked Petri net is a bipartite graph, $\mathcal{M} = (P, T, A, m_0)$, where P, T and A are a set of places, transitions and arcs, respectively, and m_0 is the initial marking function.
- A transition, t , is *enabled* in marking m iff $\forall p \in \text{Inp}(t) : m(p) > 0$. An enabled transition can *fire* — this removes one token from each of the transition's input places and adds one token to each of its output places.
- The *reachability graph* of a marked net can be derived by exhaustively determining all possible markings.
- When no transition is enabled, the net is *deadlocked*. The *firing sequence* t_2, t_4, t_1, t_2, t_4 results in the net above becoming deadlocked.

Petri Net Interface Model

A model of a component's interface is a labelled Petri net:

$$\mathcal{M}_i = (P_i, T_i, A_i, L_i, \ell_i, m_i, F_i)$$

where (P_i, T_i, A_i, m_i) is a deadlock-free, marked Petri net, L_i is an alphabet representing a set of services which are associated with transitions by a labelling function $\ell_i : T_i \rightarrow L_i \cup \{\varepsilon\}$, where ε is the empty label, $\varepsilon \notin L_i$, and F_i is a set of *final markings*, $F_i \subseteq M(\mathcal{M}_i)$.

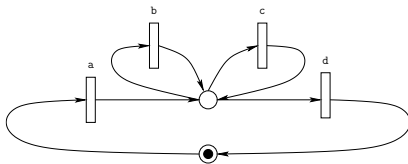


Component interactions occur between requester interfaces (*r-interfaces*) and provider interfaces (*p-interfaces*). The same component may have several r-interfaces and several p-interfaces.

Interface Languages

The language of $\mathcal{M}_i = (P_i, T_i, A_i, L_i, \ell_i, m_i, F_i)$, denoted by $\mathcal{L}(\mathcal{M}_i)$, is the set of all strings over L_i obtained by labelling firing sequences which begin with m_i and end at one of the final markings.

For example, for the previous interface model:



If $F_i = \{m_i\}$, then the language describing the behaviour of this interface is:

$$(a(b|c)^*d)^*.$$

Petri nets can represent all regular, some context-free and even some context-sensitive languages.

Component Compatibility

- In order for two components to be compatible, any sequence of services that the requester may demand must be satisfied by the provider:

Interface models of requester and provider, \mathcal{M}_i and \mathcal{M}_j , respectively, are *compatible* iff

$$\mathcal{L}(\mathcal{M}_i) \subseteq \mathcal{L}(\mathcal{M}_j).$$

- If both provider and requester languages are regular, then the interface nets can be converted to deterministic finite automata. The compatibility property can be confirmed through using the product construction technique.

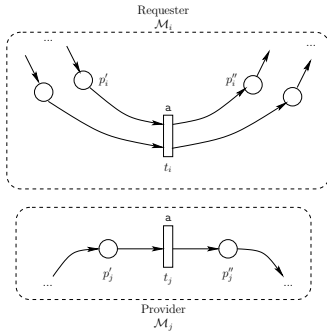
$$\mathcal{L}(\mathcal{M}_i) \cap \overline{\mathcal{L}(\mathcal{M}_j)} = \emptyset \Leftrightarrow \mathcal{L}(\mathcal{M}_i) \subseteq \mathcal{L}(\mathcal{M}_j).$$

- If one or both of the languages are non-regular, a formal means of composing two interfaces using the Petri net models must be developed.

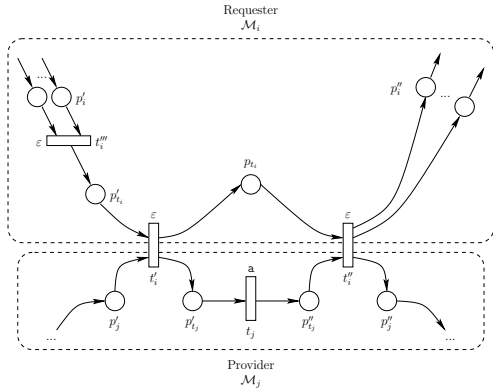
Interface Composition

The composition of two interfaces, is performed by the following transformation at their common service points:

Before



After



Interface Composition — Formally

Definition: Let $P_i \cap P_j = T_i \cap T_j = \emptyset$. A composition of an r-interface $\mathcal{M}_i = (P_i, T_i, A_i, L, \ell_i, m_i, F_i)$ and a p-interface $\mathcal{M}_j = (P_j, T_j, A_j, L, \ell_j, m_j, F_j)$, denoted $\mathcal{M}_i \triangleright \mathcal{M}_j$, is a net $\mathcal{M}_{ij} = (P_{ij}, T_{ij}, A_{ij}, L, \ell_{ij}, m_{ij}, F_{ij})$ where:

$$\begin{aligned}
 P_{ij} &= P_i \cup P_j \cup \{ p_{t_i}, p'_{t_i} : t_i \in \hat{T}_i \} \cup \{ p'_{t_j}, p''_{t_j} : t_j \in \hat{T}_j \}; \\
 T_{ij} &= T_i \cup T_j - \hat{T}_i \cup \{ t'_i, t''_i, t'''_i : t_i \in \hat{T}_i \}; \\
 A_{ij} &= A_i \cup A_j - P_i \times \hat{T}_i - \hat{T}_i \times P_i - P_j \times \hat{T}_j - \hat{T}_j \times P_j \cup \\
 &\quad \{ (p'_i, t'''_i), (t'''_i, p'_i), (p'_{t_i}, t'_i), (t'_i, p_{t_i}), (p_{t_i}, t'_i), (t'_i, p''_i) : \\
 &\quad \quad t_i \in \hat{T}_i \wedge p'_i \in \text{Inp}(t_i) \wedge p''_i \in \text{Out}(t_i) \} \cup \\
 &\quad \{ (p'_j, t'_i), (t'_i, p'_j), (p'_{t_j}, t_j), (t_j, p'_{t_j}), (p'_{t_j}, t'_i), (t'_i, p'_j) : \\
 &\quad \quad t_i \in \hat{T}_i \wedge t_j \in \hat{T}_j \wedge \ell_j(t_j) = \ell_i(t_i) \wedge \\
 &\quad \quad p'_j \in \text{Inp}(t_j) \wedge p''_j \in \text{Out}(t_j) \}; \\
 \forall t \in T_{ij} : \ell_{ij}(t) &= \begin{cases} \ell_i(t), & \text{if } t \in T_i, \\ \ell_j(t), & \text{if } t \in T_j, \\ \varepsilon, & \text{otherwise;} \end{cases} \\
 \forall p \in P_{ij} : m_{ij}(p) &= \begin{cases} m_i(p), & \text{if } p \in P_i, \\ m_j(p), & \text{if } p \in P_j, \\ 0, & \text{otherwise;} \end{cases} \\
 F_{ij} &= \{ m_{ij} : P_{ij} \rightarrow \{0, 1, \dots\} \mid (m_{ij} \downharpoonright P_i) \in F_i \wedge (m_{ij} \downharpoonright P_j) \in F_j \wedge \\
 &\quad \forall p \in P_{ij} - P_i - P_j : m_{ij}(p) = 0 \}.
 \end{aligned}$$

Component Compatibility Verification

Whether or not two components are compatible can be determined by testing the composed net for deadlock:

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

Two ways of testing for deadlock:

- Reachability analysis:
 - Systematically derive all possible markings reachable from the initial marking; dead markings can be identified in this set of markings.
 - Not suitable for unbounded nets (*i.e.*, nets in which the number of tokens is unlimited).
- Structural analysis combined with linear programming:
 - Identify net substructures called *siphons* and use linear programming to check if the number of tokens in the siphons can be minimized to zero.
 - If a net is deadlocked, then all unmarked places constitute a siphon.

Strategies for Verifying Compatibility

Reachability analysis:

Reachability analysis systematically derives all possible markings reachable from the initial marking. Dead markings can be easily identified in this set of markings.

Limitations:

- Not suitable for unbounded nets (*i.e.*, nets in which the number of tokens is unlimited) since the reachability graph becomes infinite.
- The number of reachable markings could be quite large for nets that exhibit concurrency.

Structural analysis and Linear programming:

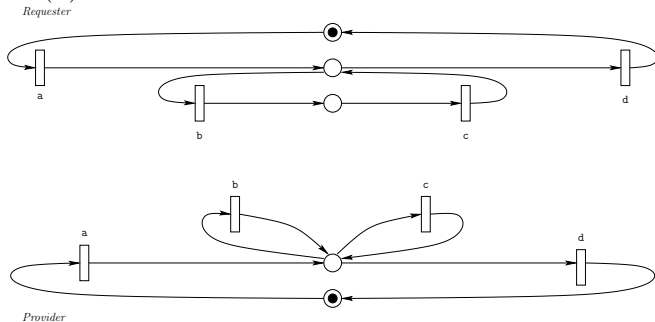
For each minimal siphon in a marked net, the siphon contains a marked trap or if the minimal number of tokens in the siphon is greater than zero, then the net is deadlock free [Xie & Chu, 1997].

Limitations:

- Finding minimal siphons can be difficult (but this may be mitigated by simplifying the composed net).
- If the number of siphons is large, many objective functions may have to be minimized during linear programming (but this number can often be reduced by simple transformations).

Example

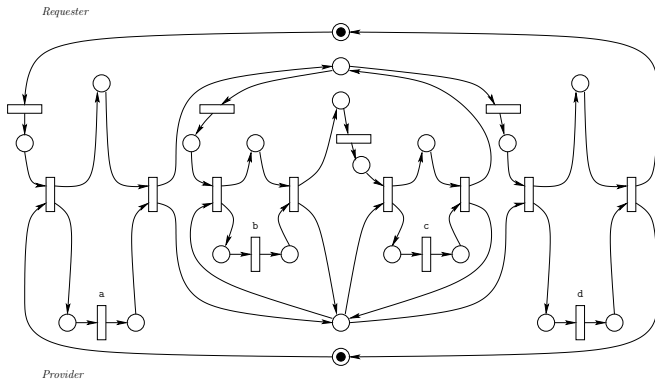
Consider database client and server components. The server (provider) supports an *open* operation (a), followed by any number of *read/write* operations in any order (b|c)* followed by a *close* operation (d).



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

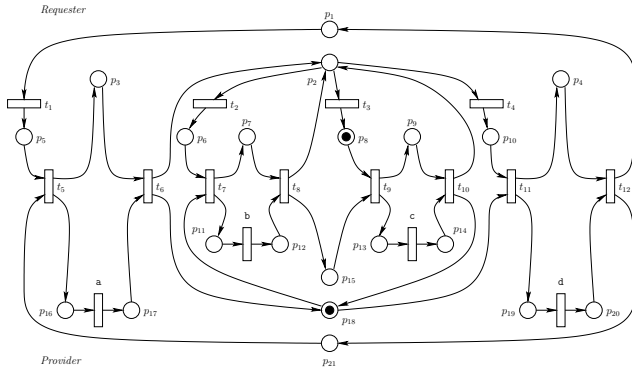
Example (cont'd)

There are 15 reachable markings, none of which result in deadlock.
(Deadlock freeness can also be verified by linear programming.) \mathcal{M}_{ij}
is deadlock-free $\Rightarrow \mathcal{M}_i$ is compatible with \mathcal{M}_j .



Example (cont'd)

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



Firing sequence resulting in deadlock: t_1, t_5, a, t_6, t_3 . This can be determined by reachability analysis or by structural analysis/linear programming.

Future Work and Open Questions

- A possible way of extracting Petri net models from high-level specifications and/or source code should be investigated.
- The composition model was initially intended for traditional software architectures. As software becomes increasingly more distributed, can this technique be applied to modern web services and/or service oriented architectures?
- Extensions of Petri nets (e.g., inhibitor arcs) provide enhanced modelling power, but introduce undecidability issues.

Concluding Remarks

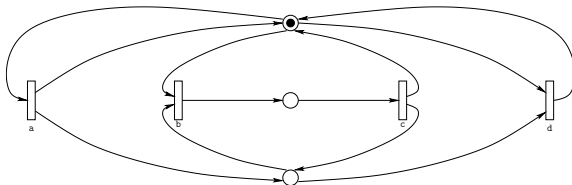
- Using appropriate definitions of *components*, *composition* and *compatibility*, this work has presented a formal model which allows one to compose components and to verify their compatibility.
- Component compatibility can be checked by representing the interface behaviours as Petri nets and then composing them.
 - If the resulting net exhibits a deadlock, the components are not compatible.
 - Deadlock detection can be done using structural properties and linear programming or reachability analysis.

Supplementary slides

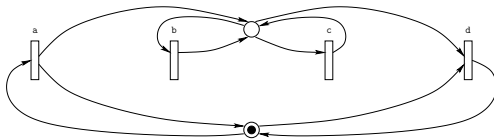
Example of Non-Regular Interface Languages

Consider the case where the open and close calls can nest in the provider and requesters:

Requester

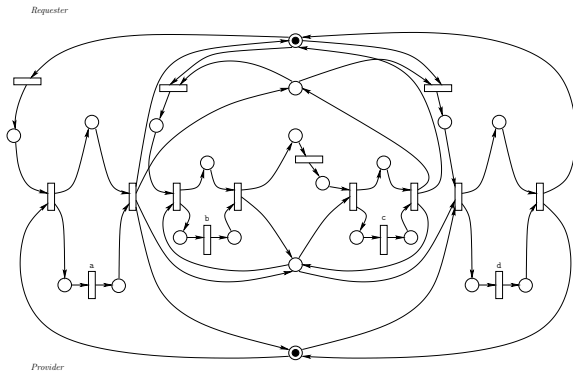


Provider



Example of Non-Regular Interface Languages (cont'd)

After composition:



It can be shown through linear programming that this net does not deadlock – the two components are compatible.