

# Verification of component behavioral compatibility

Donald Craig    Wlodek Zuberek

<sup>1</sup>Department of Computer Science  
Memorial University of Newfoundland  
Canada

<sup>2</sup>Department of Computer Science  
Memorial University of Newfoundland  
Canada

Second International Conference on  
Dependability of Computer Systems  
Szklarska Poreba, Poland, June 14-16, 2007

# 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*.

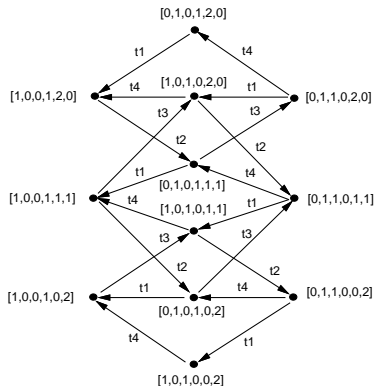
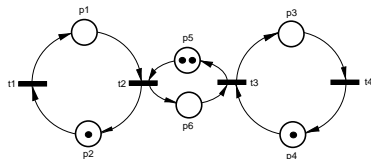
- “A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties.”

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 behaviors.

# 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,  $m_0 : P \rightarrow \{0, 1, \dots\}$ .
- A transition,  $t$ , is *enabled* by 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 exploring all possible markings.
- When no transition is enabled by a marking, the net is *deadlocked*.

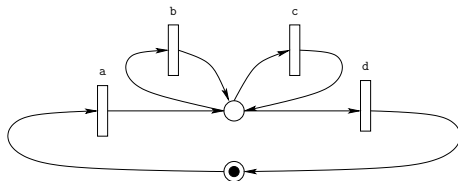


## Petri Net Interface Model

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 (cyclic) 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  $\varepsilon$  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. Provider models have some restrictions reflecting their reactive nature (unique service labels,  $\varepsilon$ -conflict-freeness).

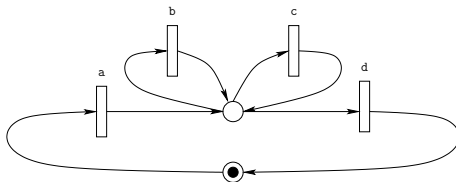
# Interface Languages

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:

$$\mathcal{L}(\mathcal{M}_i) = \{ \ell(\sigma) \mid \sigma \in \mathcal{F}(\mathcal{M}_i) \wedge m_i \xrightarrow{\sigma} m \wedge m \in F_i \}$$

where  $\ell(t_{i_1} \dots t_{i_k}) = \ell(t_{i_1}) \dots \ell(t_{i_k})$ .

For example, for the previous interface model:



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

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

# Component Compatibility

- In order for two components to be compatible, **the set** of services required by the requester,  $L_i$  must be made available by the provider,  $L_i$ , *i.e.*,

$$L_i \subseteq L_j.$$

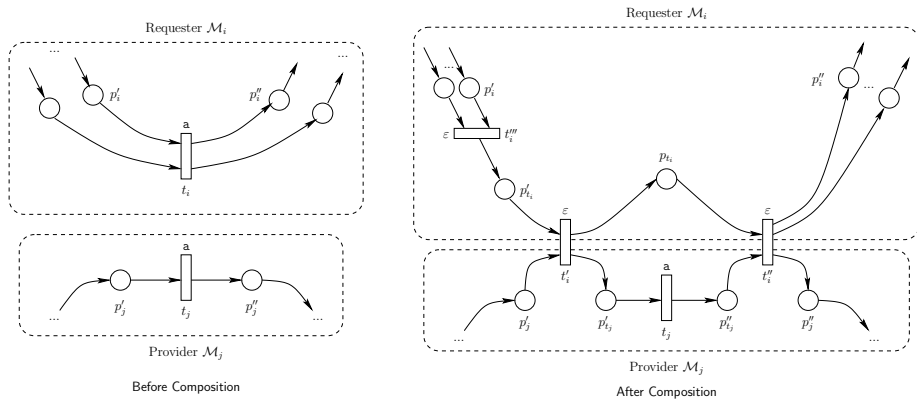
- In addition, **any sequence** of services that the requester may demand must be satisfied by the provider.
- Interface models of requester and provider,  $\mathcal{M}_i$  and  $\mathcal{M}_j$ , respectively, are *compatible* iff

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

- Compatibility relation can be verified by analyzing the interacting Petri net interface models.
- Models of interacting components are obtained by composing requester and provider models. Such a composition can be performed in several ways, creating nets with different properties. In the proposed composition, a requester can affect the behavior of a provider, but a provider cannot affect the behavior of its requester.

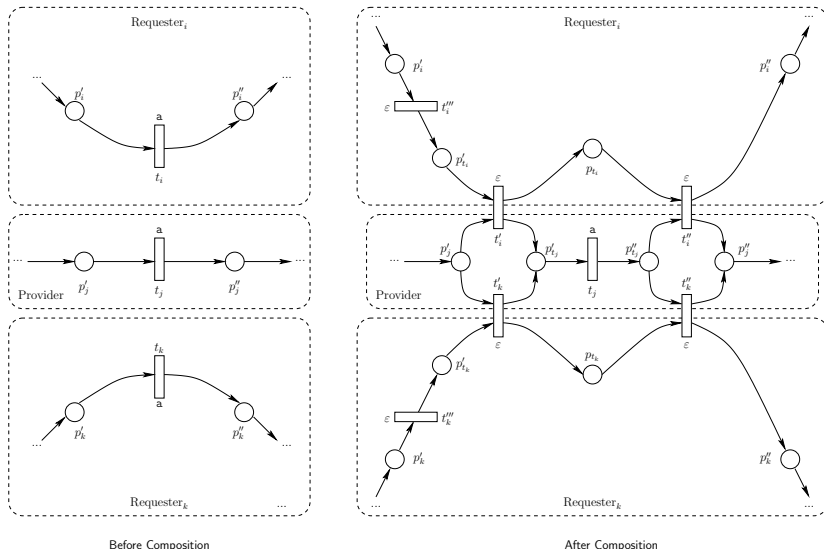
# Interface Composition

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



# Multiple Requesters, Single Provider

Multiple requesters are composed with a single provider in a similar way:



Before Composition

After Composition



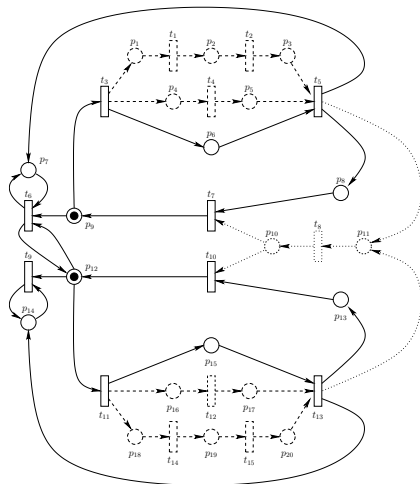
# Component Compatibility Verification

- For the proposed composition, component incompatibilities results in deadlocks; component compatibility verification is thus performed by deadlock analysis of the composed model.
- Testing for deadlock can use structural analysis combined with linear programming. Other approaches to deadlock detection include reachability analysis, net unfolding, etc.
- Structural analysis identifies net substructures called *siphons* and uses linear programming to check if tokens can be removed from siphons.
- A siphon is a subset of places,  $S$ , such that  $\text{Inp}(S) \subseteq \text{Out}(S)$ . Minimal siphons are siphons which do not contain other siphons. Basis siphons are siphons from which all other siphons can be obtained by the union operation.
- If a net is deadlocked, then all unmarked places constitute a siphon. This siphon is composed of (one or more) unmarked basis siphons and (one or more) unmarked minimal siphons.

# Equivalent siphons

- The drawback of siphon-based analysis is that in some net models the number of (basis) siphons can be quite large.
- Siphon equivalence can be used to reduce this number. Two siphons are equivalent iff for each marking reachable from the initial marking either both siphons are marked or are both are unmarked.
- For deadlock analysis, only one siphon from each equivalence class of (basis and minimal) siphons is needed.
- Elimination of equivalent siphons preserves the existence (or absence) of deadlocks.
- Parallel and alternate paths introduce equivalent siphons.

# Parallel and Alternate Paths



*original net*

76 proper basis siphons

15 proper minimal siphons

*reduced net*

2 proper basis siphons

2 proper minimal siphons

## Deadlock Detection

The invocation is  $deadlock(m_0, S_m, S_b)$ , where  $S_m$  is the (reduced) set of minimal siphons and  $S_b$  is the (reduced) set of basis siphons:

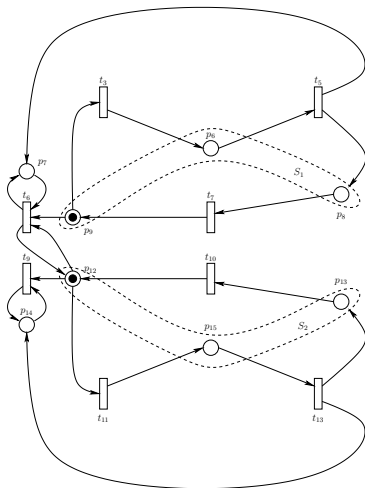
```
function  $deadlock(m, X, Y)$  : boolean;  
begin  
  if  $enable(m) = \emptyset$  then return TRUE fi;  
  if  $X \neq \emptyset$  then  
    for each  $x$  in  $X$  do  
       $(v, n) := LPminimize(x, m)$ ;  
      if  $nonzero(v) \wedge n = 0 \wedge feasible(v, m)$  then  
         $m' := m + \mathbf{C} \times v$ ;  
         $X' := marked(X, m')$ ;  
        if  $deadlock(m', X', Y)$  then return TRUE fi  
      fi  
    do  
  fi;  
  if  $Y \neq \emptyset$  then return  $deadlock(m, Y, \emptyset)$  fi;  
  return FALSE  
end
```

## Feasibility Check

The linear programming procedure returns a *firing vector* minimizing the number of tokens in the analyzed siphon. Such a firing vector may have no implementation in the form of a *firing sequence*, i.e., it may be infeasible for a given marking  $m$ . Therefore the feasibility of firing vectors is checked by the following recursive (boolean) function:

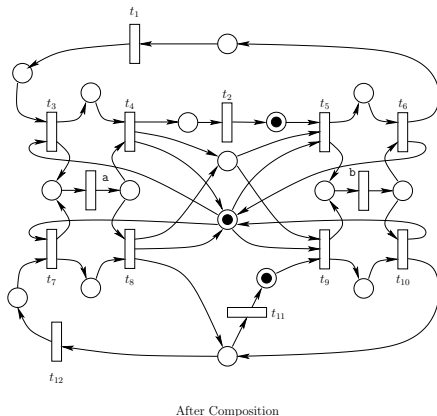
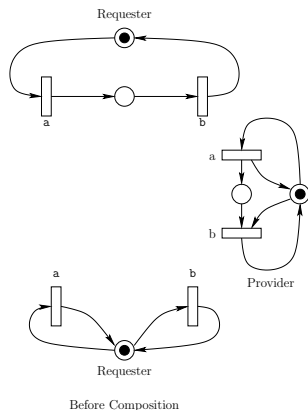
```
function feasible( $v, m$ ) : boolean;  
begin  
  if zero( $v$ ) then return TRUE fi;  
  for each  $t$  in enable( $m$ ) do  
    if  $v[t] > 0$  then  
       $v' := v$ ;  
       $v'[t] := v'[t] - 1$ ;  
       $m' := \text{fire}(m, t)$ ;  
      if feasible( $v', m'$ ) then return TRUE fi  
    fi  
  od;  
  return FALSE  
end
```

# Deadlock Detection



siphon  $S_1$  becomes unmarked by sequence  $\{t_3, t_5, t_7, t_6\}$ , and then siphon  $S_2$  becomes unmarked by sequence  $\{t_{11}, t_{13}, t_{10}, t_9\}$ .

# Single-Provider/Multi-Requester Example



- Reducing the net by eliminating several equivalent siphons results in a net which has seven basis siphons, one of which is minimal.
- The firing sequence that results in the dead marking shown in the composed net is:  $(t_1, t_3, a, t_4, t_2, t_{11}, t_9, b, t_{10}, t_{11})$ .

# 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 other techniques developed for deadlock detection.
- Component compatibility can be used in integrating software systems from independently developed components, in substitutability analysis, and other aspects of component-based software development.
- Although open questions remain (i.e., how to get interface models), it is believed that the proposed approach constitutes an interesting contribution to the area of software design and development.