Petri Nets for Services Modeling and Composition

Dr. Dmytro Zhovtobryukh
Department of Mathematical Information Technology

March 11 and 13 2008
Visiting lecture for the TJTSE54 “Development Methods and Architectures in Commerce” course
Outline

I Services and SOA
II Petri Nets
III Services Modeling
IV Service Composition
V Automated Service Composition
I Services and SOA
Concepts

What is a service?
- a service is the non-material equivalent of a good (economics and marketing)
- a discrete business or technical function (enterprise and system architectures)
- an (autonomous) software system
- A service is a process!

What is a process?
- is a naturally occurring or designed sequence of changes of properties or attributes of a system or an object
Service Oriented Computing

**Service Oriented Computing (SOC)** is the computing paradigm that utilizes services as fundamental elements for developing applications [Papazoglou and Georgakopoulos, 2003]

SOC relies on the Service Oriented Architecture (SOA), which is a way of reorganizing software applications and infrastructure into a set of interacting services [Papazoglou, 2003]

**Service Oriented Architecture** is a paradigm for organizing and utilizing distributed capabilities that may be under the control of different ownership domains [OASIS’s SOA reference specification, 2006]
SOA Services

- A service is a contractually defined **behavior** that can be implemented and provided by a component for use by another component [Nickull, 2005]
- Services are self-describing, open components that support rapid, low-cost **composition** of distributed applications [Papazoglou and Georgakopoulos, 2003]
- SOA implementations:
  - Web Services
  - J2EE
  - .NET
SOA Services (2)

- A Web Service is a software system designed to support interoperable machine-to-machine interaction over a network (Web Service Architecture, W3C Note, 2004)

- Properties:
  - Technological neutrality
  - Loose coupling with clients
  - Transparency (location, access, …)
  - Flexible configurability

- Major outcome: Interoperability!
Service Oriented Architecture (SOA)

- 3 basic parties (roles):
  - Service Provider
  - Service Consumer
  - Service Broker (Registry)

- 3 basic operations (interactions):
  - Publish/Unpublish
  - Find
  - Bind

- 3 native capabilities:
  - Communication (SOAP)
  - Description (WSDL)
  - Discovery (UDDI)

Service Description:
- Capability
- Interface
- Behavior (process)
- QoS
Extended SOA

- 2 more layers:
  - Composition
  - Management
- 2 more roles:
  - Service Aggregator
  - Service Operator
- Composite service capabilities:
  - Coordination
  - Conformance
  - Monitoring
  - QoS
II Petri Nets
What are Petri Nets?

Petri Nets is a formal tool for study and modeling of systems and processes. *Petri nets is a well-founded process modeling technique that has formal semantics* [Peterson, 1981]

Emerged with the dissertation of Carl Adam Petri in 1962

Petri nets are basically used in two ways:

- Refinement of system’s design through external Petri net model analysis
- System development and analysis on the basis of core Petri net model
Benefits of Petri Nets

- Formality and abstract representation
- Vivid graphical representation of models
- Focus on interactions and dynamic relationships between systems’ or processes’ elements
- Capturing causality between events in complex processes
- Modeling of concurrency and non-determinism
- Rich set of formal tools for model analysis
Petri Nets Basics

- Petri nets are formally defined and manipulated in terms of sets (bags).
- A Petri net is a four-tuple $C=(P,T,I,O)$, where:
  - $P$ is a finite set of places $P = \{p_1, p_2, \ldots, p_n\}$
  - $T$ is a finite set of transitions $T = \{t_1, t_2, \ldots, t_m\}$
  - The set of places and the set of transitions are disjoint. $P \cap T = \emptyset$
  - $I$ is the input function, a mapping from transitions to bags of places. $I : T \rightarrow P^\infty$
  - $O$ is the output function; a mapping from transitions to bags of places. $O : T \rightarrow P^\infty$
Graphical Representation

A Petri net is a bipartite directed multigraph \( G = (V, A) \)
- \( V \) is a set of vertices
  \( V = \{v_1, v_2, \ldots, v_s\} \)
- \( A \) is a bag of directed arcs
  \( A = \{a_1, a_2, \ldots, a_r\} \)
- \( V \) is partitioned into two disjoint sets \( P \) and \( T \) such that
  \( V = P \cup T \), \( P \cap T = \emptyset \)
- For each directed arc \( a_i = (v_j, v_k) \), either \( v_j \in P, v_k \in T \)
  or \( v_j \in T, v_k \in P \)
**Organisation Principles**

- Bipartition: places can be connected to transitions and vice versa, but not places to places or transitions to transitions.
- Arcs are always directed.
- For every transition, multiple input/output places are possible.
- One place can be both input and output for the same transition.
- A place can be a multiple input or a multiple output of a transition.
Petri Net Markings and Execution

- Markings are representing states of a Petri net.
- Tokens are used to distinguish between markings.
- Tokens reside in places (multiple in general case).
- A marking $\mu$ of a Petri net $C=(P,T,I,O)$ is an $n$-vector
  $\mu = \{\mu_1, \mu_2, \ldots, \mu_n\}$
  where $n = |P|$
- Petri net is executed by firing transitions.
- A transition should be **enabled** in order to be fired.
Petri Net Markings and Execution (2)

- A transition is enabled whenever each of its input places contain as many tokens as the number of arcs connecting this place to the transition.
- On firing the transition tokens are removed from all its input places and deposited into all its output places (one token per arc).
- Transition firings change Petri net markings, i.e., change its states.
- Petri net execution persists until a deadlock marking is reached, i.e., a marking in which no transitions can be any more fired.
Petri Nets Classes

- Basic class: Place/Transition nets or ordinary Petri nets

- Restrictions
  - state machines
  - marked graphs
  - free-choice nets
  - simple nets

- Extensions
  - Increasing modeling power
    - Inhibitor arcs
    - Disjunctive transitions
    - Switch Transitions
  - Modifying execution rules
    - Coloured Petri nets
    - Timed Petri nets
    - Stochastic Petri nets
    - Prioritized Petri nets
    - Meta Petri nets
III Service Modeling
Web Services are modeled as processes exhibiting concurrency and non-determinism.

Web Service process (execution flow) is a partially ordered set of service states and service operations.

A state of a Web Service is a unique set of attributes’ values that describe the process (flow of a Web Service) at any given point in time.

A Web Service operation is an action which switches the service from one state to another. Each operation has at least one input state (prerequisite) and at least one output state (effect). A service operation can be performed whenever its input state(s) is(are) active.
Concepts (2)

- An operation can be understood as a service function which changes the state of the service.
- A service always has an initial state (at which it starts) and a final state (at which it terminates).
- Service execution is a sequence of service operations which leads from the service’s initial state to its final state.
- Service can be treated as an operation since it basically encapsulates a number of other operations.
- Structural types of services:
  - Atomic (a single service operation)
  - Simple (multiple service operations)
  - Composite (a component service operation is a service itself)
Modeling Options

- What should be modeled?
  - Control flows (temporal, causal relationships)
  - Data flows (functional dependencies)
  - Resource flows (physical constraints)

- Modeling techniques
  - Orchestration languages (e.g., BPEL4WS)
  - $\pi$-Calculus
  - Petri nets
  - Finite State Machines
  - UML
  - …
**Web Services as Petri Nets**

- *Service net* is a Petri net model of a service
- Places of a service net model states of a service
- Transitions model service operations
- Service nets are Petri nets in *standard form*, i.e., they have exactly one initial place and one final place
- A service net is a tuple $SN=(P,T,F,i,o)$, where
  - $P$ is a finite set of places
  - $T$ is a finite set of transitions, such that $P \cap T = \emptyset$
  - $F$ is a flow relation $F \subseteq (P \times T) \cup (T \times P)$
  - $i$ is the input place that has no incoming arcs
  - $o$ is the output place that has no outgoing arcs
- The initial marking $M_0=\{1,0,\ldots,0\}$
Basic Constructs

- Empty service
  - Performs no operation
  - No state change
  - Introduced for theoretical reasons

- Atomic service
  - Performs single (i.e., non-decomposable) operation
  - Has no intermediate states
  - Indivisible

![Empty Service Diagram](image1)

![Atomic Service Diagram](image2)
Simple Service

- Contains multiple operations and/or operations decomposable into more primitive ones
- Has a number of intermediate states
- Can be decomposed into multiple services
- May exhibit concurrency and/or non-determinism

Simple Service

- Get coordinates of A
- Calculate $D_X = X_A - X_B$
- Calculate $D_Y = (D_Y)^2$
- Get coordinates of B
- Compute Cartesian coordinates of A
- Calculate $D' = D_X^2 + D_Y^2$
- Compute Cartesian coordinates of B
- Calculate $D = D' + D'$

Visiting lecture "Petri Nets for Services Modeling and Composition"
Composite Services

- A composite service is a service which comprises other service(s)

- A composite service generally consists of:
  - Own service states (at least two)
  - Own service operations (optional)
  - Component services (at least one)
  - Auxiliary linkage functionality (places and transitions)

- Two classes of composite services
  - Purely composite services (no exclusive service operations)
  - Partially composite services (own service operations present)
IV Service Composition
Service Composition Algebra

- Service composition can be performed using declarative expressions.
- Composition declarations are built using specific service composition algebra.
- Service composition algebra provides means to formalize a variety of causal and/or functional relationships between components of a composite service.
- A composite service’s control flow can be uniquely and unambiguously represented by a corresponding expression written in service composition algebra.
- Each service expression takes component services as operands and connect them using a number of composition operators.
Service Composition Operators

- Service composition algebra includes the following basic composition operators:
  - Sequence
  - Mutual exclusion
  - Parallelism
  - Iteration

- Composition algebra verifies closure property, i.e., all results of utilization of algebraic operators on valid services are valid services.

- Any algebraic operator as well as any expression written in service composition algebra has an equivalent Petri net representation and vice versa.
Service Composition Operators (2)

- Service composition algebra may also include advanced operators of the following types:
  - Combination of basic operators with reduction of composition complexity (e.g., unordered sequence)
  - Extension of basic operators with additional features (e.g., parallelism with communication)
  - Special cases modeling operators (e.g., refinement)
- The set of advanced operators is not fixed. It can be extended with new operators should such prove they satisfy algebra’s closure property
Sequence

- Binary operator
- Executes two services in sequence, i.e., one after another
- Models causality
  - logical causality
  - functional causality
- Typical use case: supply chain
- Non-commutative:
  \[ S_1 \triangleright S_2 \neq S_2 \triangleright S_1 \]
- Associative:
  \[ (S_1 \triangleright S_2) \triangleright S_3 = S_1 \triangleright (S_2 \triangleright S_3) \]

\[ S = S_1 \triangleright S_2 \]
Mutual Exclusion

- Binary operator
- Executes two services alternatively, i.e., either one or the other
- Models non-determinism (using conflict split/merge): the choice between services by default is made randomly
- Typical use cases:
  - alternate functionality
  - fault tolerance

\[ S = S_1 \otimes S_2 \]

- Commutative:
  \[ S_1 \otimes S_2 = S_2 \otimes S_1 \]

- Associative:
  \[ (S_1 \otimes S_2) \otimes S_3 = S_1 \otimes (S_2 \otimes S_3) \]
Parallelism

- Binary operator
- Executes two services concurrently, i.e., both, simultaneously and in parallel
- Models concurrency (using concurrency split/merge)
- Typical use cases:
  - causally independent services (for performance increase)
  - mutually dependent services (functional correlation)
  - testing

\[ S = S_1 + S_2 \]

Commutative:
\[ S_1 + S_2 = S_2 + S_1 \]

Associative:
\[ (S_1 + S_2) + S_3 = S_1 + (S_2 + S_3) \]
Iteration

- Unary operator
- Cyclic execution of a service, i.e., a service is executed certain number of times in a row
- Models finite cycling or conditional cycling (using extra control place and an inhibitor arc)
- Typical use cases:
  - conveyor (finite iteration)
  - state maintenance (conditional iteration)
Unordered Sequence

- Binary operator
- Advanced operator type 1
- Executes two services in sequence of arbitrary order, i.e., either S2 after S1 or vice versa
- Typical use cases:
  - independent services (performance issues) – alternative to parallelism
  - complementary services (logical correlation)
- Commutative: \( S_1 \leftrightarrow S_2 = S_2 \leftrightarrow S_1 \)
- Non-associative: 
  \[
  (S_1 \leftrightarrow S_2) \neq S_1 \leftrightarrow (S_2 \leftrightarrow S_3)
  \]

\[
S = S_1 \iff S_2 = (S_1 \triangleright S_2) \otimes (S_2 \triangleright S_1)
\]
Parallelism with Communication

- Binary operator
- Advanced operator type 2
- Executes two services in parallel, but execution of individual operations is synchronized between concurrent services
- Parallelism is a special case of this operator
- Communication set $C$ is used to formalize synchronization facility
- Typical use case:
  - Mutually dependent services (functional correlation)

Mathematical representation:

\[ S = S_1 + S_2 \]

Commutative:

\[ S_1 + S_2 = S_2 + S_1 \]

Structurally associative
Refinement

- Binary operator
- Advanced operator type 3
- Substitutes an individual operation (or a group of operations) of a service with a standalone service
- Services execution is nested
- Especially useful for modeling of partially composite service
- Non-commutative: \[ S_1 \triangledown S_2 \neq S_2 \triangledown S_1 \]
- Associative: \[ (S_1 \triangledown S_2) \triangledown S_3 = S_1 \triangledown (S_2 \triangledown S_3) \]

\[ S = S_1 \triangledown S_2 \]
Example

\[
\left( S_1 \nabla S_2 + S_3 \right) \triangleright \left( S_4 \otimes S_5 \right) \triangleright S_6
\]
Service Modeling from Scratch

- Made through preliminary planning which results in creation of a *generic service*
- Component services are inserted into generic service plan instead of its generic operations using refinement operator
- Modeling from scratch is usually applied for creation of purely composite services
Hierarchical modeling is a process of gradual refinement of services with other services. Component services are nested into component services of a higher level. Helps to increase quality of a service and bring additional features into it. Particularly useful for creation of partially composite services (re-planning).
Algebraic Properties

Algebraic properties are rules for manipulating the structure of composite services.

- **Commutativity properties** indicate arbitrariness of operands order with the corresponding operators.
- **Associativity properties** indicate arbitrariness of operators' execution order for two or more consecutive operators of the same type.
- **Distributivity properties** are the most powerful tool of composition complexity reduction and structure optimization.

### Algebraic Properties

<table>
<thead>
<tr>
<th>Property</th>
<th>Example</th>
<th>Rule</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Commutativity</strong></td>
<td>( S_1 \circ S_2 = S_2 \circ S_1 )</td>
<td>(3)</td>
</tr>
<tr>
<td></td>
<td>( S_1 \oplus S_2 = S_2 \oplus S_1 )</td>
<td>(4)</td>
</tr>
<tr>
<td></td>
<td>( S_1 + S_2 = S_2 + S_1 )</td>
<td>(5)</td>
</tr>
<tr>
<td><strong>Associativity</strong></td>
<td>( (S_1 \circ S_2) \circ S_3 = S_1 \circ (S_2 \circ S_3) )</td>
<td>(6)</td>
</tr>
<tr>
<td></td>
<td>( (S_1 \oplus S_2) \oplus S_3 = S_1 \oplus (S_2 \oplus S_3) )</td>
<td>(7)</td>
</tr>
<tr>
<td></td>
<td>( (S_1 + S_2) + S_3 = S_1 + (S_2 + S_3) )</td>
<td>(8)</td>
</tr>
<tr>
<td><strong>Distributivity</strong></td>
<td>( (S_1 \circ S_2) \circ S_3 = (S_1 \circ S_3) \circ (S_2 \circ S_3) )</td>
<td>(9)</td>
</tr>
<tr>
<td></td>
<td>( (S_1 \oplus S_2) \oplus S_3 = (S_1 \oplus S_3) \oplus (S_2 \oplus S_3) )</td>
<td>(10)</td>
</tr>
<tr>
<td></td>
<td>( (S_1 + S_2) + S_3 = (S_1 + S_3) + (S_2 + S_3) )</td>
<td>(11)</td>
</tr>
<tr>
<td><strong>Idempotence</strong></td>
<td>( S \circ S = S )</td>
<td>(12)</td>
</tr>
<tr>
<td></td>
<td>( S \oplus S = S )</td>
<td>(13)</td>
</tr>
<tr>
<td></td>
<td>( S + S = S )</td>
<td>(14)</td>
</tr>
<tr>
<td><strong>Identity</strong></td>
<td>( S \circ S = S )</td>
<td>(15)</td>
</tr>
<tr>
<td></td>
<td>( S \oplus S = S )</td>
<td>(16)</td>
</tr>
<tr>
<td></td>
<td>( S + S = S )</td>
<td>(17)</td>
</tr>
<tr>
<td><strong>Identity</strong></td>
<td>( S \circ S = S )</td>
<td>(18)</td>
</tr>
<tr>
<td></td>
<td>( S \oplus S = S )</td>
<td>(19)</td>
</tr>
<tr>
<td></td>
<td>( S + S = S )</td>
<td>(20)</td>
</tr>
<tr>
<td><strong>Commutativity</strong></td>
<td>( S_1 \circ S_2 = S_2 \circ S_1 )</td>
<td>(21)</td>
</tr>
<tr>
<td></td>
<td>( S_1 \oplus S_2 = S_2 \oplus S_1 )</td>
<td>(22)</td>
</tr>
<tr>
<td></td>
<td>( S_1 + S_2 = S_2 + S_1 )</td>
<td>(23)</td>
</tr>
<tr>
<td><strong>Distributivity</strong></td>
<td>( (S_1 \circ S_2) \circ S_3 = (S_1 \circ S_3) \circ (S_2 \circ S_3) )</td>
<td>(24)</td>
</tr>
<tr>
<td></td>
<td>( (S_1 \oplus S_2) \oplus S_3 = (S_1 \oplus S_3) \oplus (S_2 \oplus S_3) )</td>
<td>(25)</td>
</tr>
<tr>
<td></td>
<td>( (S_1 + S_2) + S_3 = (S_1 + S_3) + (S_2 + S_3) )</td>
<td>(26)</td>
</tr>
<tr>
<td><strong>Idempotence</strong></td>
<td>( S \circ S = S )</td>
<td>(27)</td>
</tr>
<tr>
<td></td>
<td>( S \oplus S = S )</td>
<td>(28)</td>
</tr>
<tr>
<td></td>
<td>( S + S = S )</td>
<td>(29)</td>
</tr>
<tr>
<td><strong>Identity</strong></td>
<td>( S \circ S = S )</td>
<td>(30)</td>
</tr>
<tr>
<td></td>
<td>( S \oplus S = S )</td>
<td>(31)</td>
</tr>
<tr>
<td></td>
<td>( S + S = S )</td>
<td>(32)</td>
</tr>
</tbody>
</table>
Web Service Analysis

- Modeling by itself is of little use without analysis.
- Analysis evaluates and validates modeling efforts, thus making modeling beneficial against direct system (process) implementation.
- Petri nets modeling formalism offers powerful formal instrumentation to conduct analysis of built Petri net models.
- Analysis of several important properties of Petri net models helps to discover flaws of service design or verify its correctness.
Safeness and Boundedness

- A place of a Petri net is *k-safe* (*k-bounded*) if the number of tokens in it never exceeds *k* for any reachable marking of the Petri net. Accordingly, the Petri net is *k*-safe when all its places are *k*-safe.
- Safeness helps verify that no livelocks are possible within the model.
- Generally, ordinary Petri nets in standard form are safe:
  - no multiple arcs
  - appropriate use of conflict split/merge constructs
- Although some of the composition operators may be unsafe, they are always bounded.
Conservation

A Petri net is conservative if the total number of tokens present in the net is constant in any reachable marking.

Conservation property is especially important for modeling resources and resource allocation systems.

Conservation verifies that tokens neither destroyed, nor created during service execution.

Parallelism operators duplicate tokens, but they successfully consume them with their output merges.

It is important to check conservation property between input and output places of composition constructs.
Liveness

Liveness ensures that a service model is accomplishable.

Liveness of a service net is determined by the liveness level of its output transition $t_0$.

- Level 0: the transition $t_0$ cannot be fired, since there is no firing sequence in any reachable marking such that this transition can be enabled. This level corresponds to a deadlock and to a dead service.
- Level 1: the transition $t_0$ is potentially fireable, i.e. there is a reachable marking in which this transition is enabled. This level corresponds to a potential deadlock and a potentially live service.
- Level 2: the transition $t_0$ can occur within firing sequence a finite number of times. This level corresponds to a finite live cycle.
- Level 3: the transition $t_0$ can occur within firing sequence infinitely often. This level corresponds to an infinite live cycle.
- Level 4: the transition $t_0$ is always fireable, i.e. for any reachable marking except for the final marking there exists a firing sequence such that this transition is enabled. This level corresponds to a strictly live service.

The necessary condition of service model’s feasibility is liveness at level 1.

The sufficient condition of service model’s feasibility is liveness at level 4.
Reachability and Reachability Trees

- Is final marking reachable from initial one?
- Reachability tree is the major analysis tool for problems of:
  - safeness
  - conservation
  - liveness
  - reachability

![Reachability Tree Diagram]

- Is final marking reachable from initial one?
- Reachability tree is the major analysis tool for problems of:
  - safeness
  - conservation
  - liveness
  - reachability
V Automated Service Composition
Motivating Scenario
What is Service Composition about?

From user perspective:
- Automation of user tasks and activities
- Customization of the value provided by services with respect to user needs

From service perspective:
- Seamless interoperation among services
- Integration of individual services’ functionality
- Reusability of services
Web Service Composition Approaches

- Manual composition
  - Web Service Orchestration/Choreography languages (e.g., BPEL4WS, WS-CDL)

- Semi-automated composition
  - Automatic service discovery and matchmaking using semantic service descriptions (e.g., OWL, WSML)

- Full-automated composition
  - Goal-driven composition (AI planning techniques)
  - Automatic control flow construction
  - User is out of the loop
Service Composition Planning Process

1. User Request
2. Goal Extraction
3. Goal Decomposition
4. Control Flow Construction
5. Service Discovery
6. Service Matchmaking
7. Control Flow Optimization and Analysis

- User Request
- Goal Extraction
- Goal Decomposition
- Control Flow Construction
- Service Discovery
- Service Matchmaking
- Control Flow Optimization and Analysis

Goal related functionality
Control flow related functionality
Service related functionality
Service net construction procedures
Data flows
Thank You!

Questions?