CSCI427/CSCI927 Service-Oriented Software Engineering

Service Change Management

Change management

- Changes are inevitable in software development
  - **New requirements** emerged at any time in the software development lifecycle.
    - E.g. new functionalities
  - Changes in **business environments**
    - E.g. competition, laws, new markets, new customers, etc.
  - Changes in **infrastructure environments**
    - E.g. new servers, new equipments, etc.
  - **New technology** arriving
    - E.g. New version of OS, new standards, etc.
  - **Bugs** need fixing
  - **Performance** needs improvement

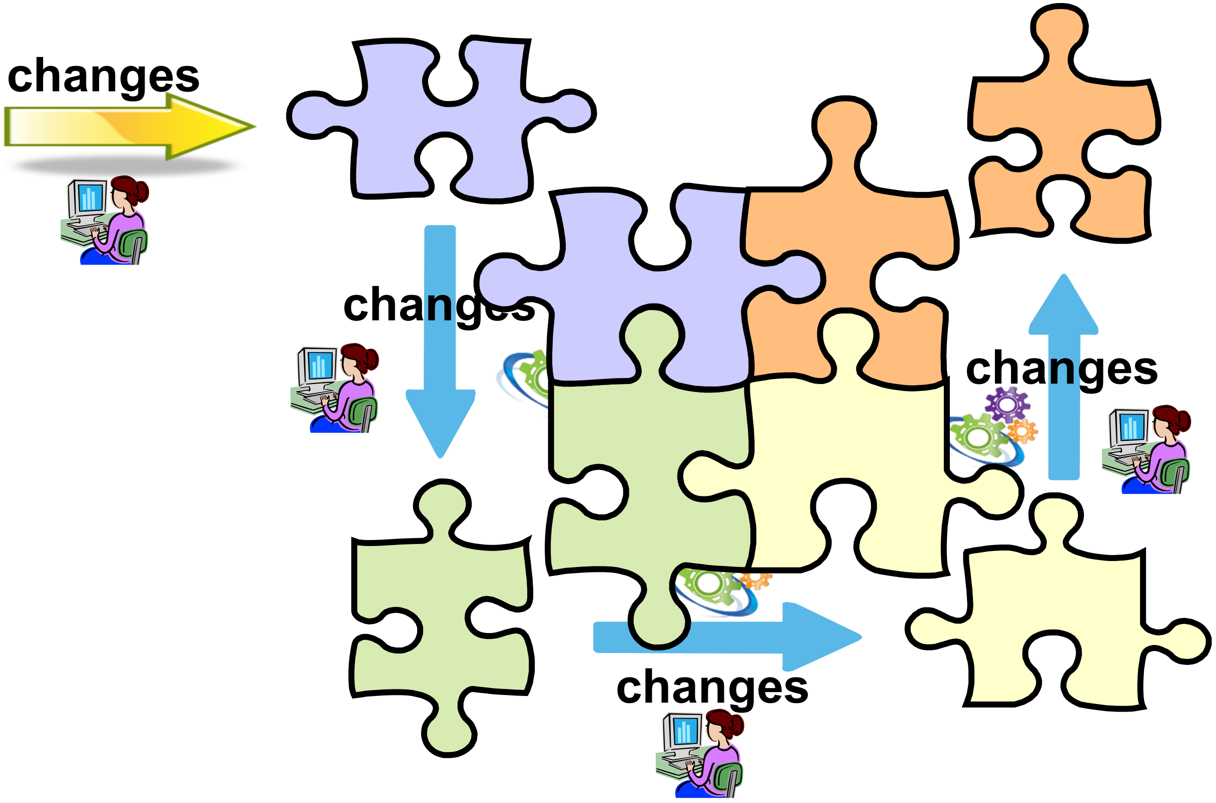
Change management (cont.)

- Change management provides a structured framework for handling both **maintenance and evolution** changes.

- Maintenance and evolution are critical, accounting for a majority of a system’s cost.
  - More than 60% of software developers will be working on software maintenance and evolution.

- As the organization grows and the business environment rapidly changes, **changes to the service-oriented architecture** (SOA) are inevitable.

Change propagation



The ripple effect

As a change is started on a software system, other coordinated changes are often needed at the same time in other parts of the software (a far-reaching consequence).

Scenario: E-commerce Website

Initial Change:

- The development team decides to update the **payment gateway API integration** in an e-commerce website.

Coordinated Changes Needed:

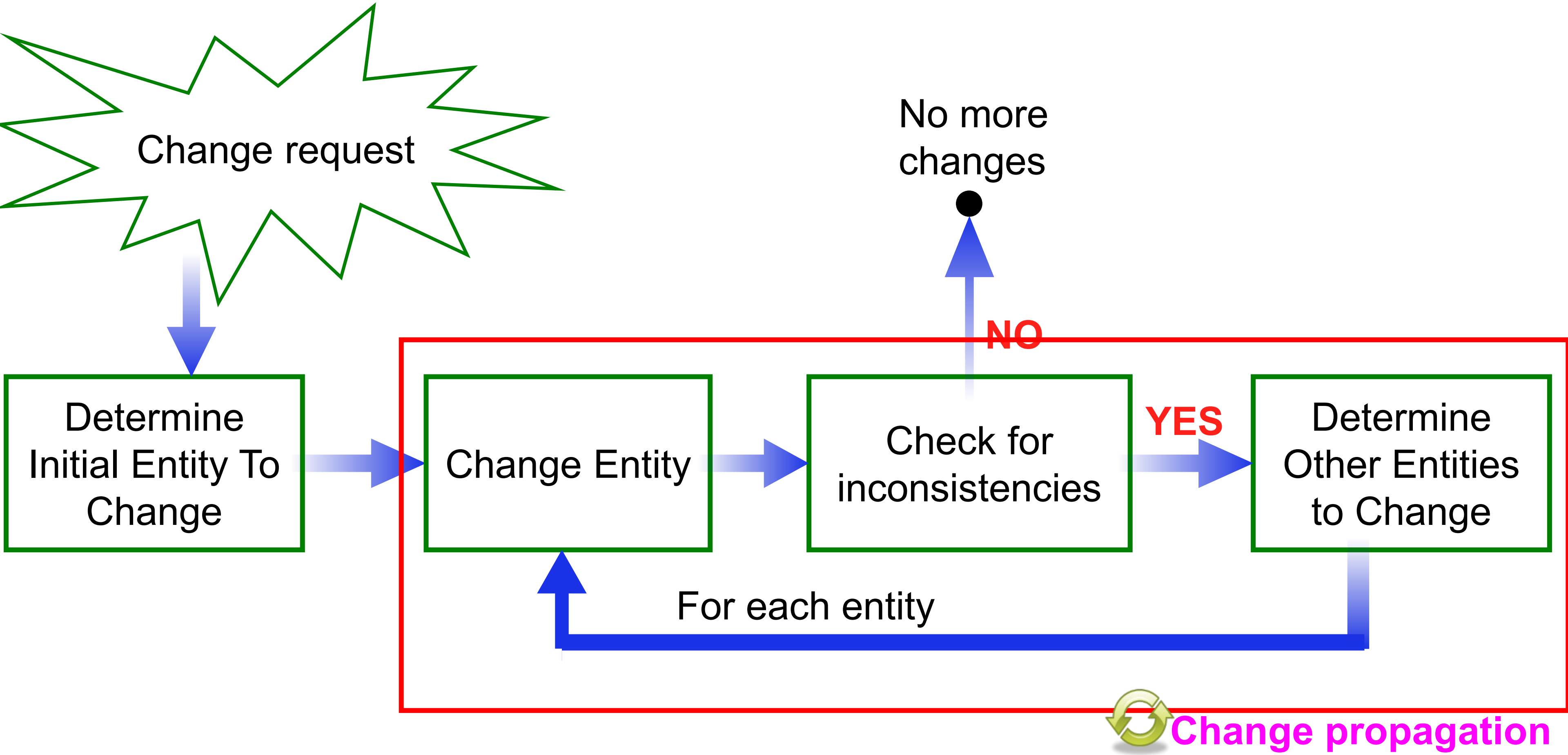
- **User Interface (UI) Update**: The new payment might support additional payment options (mobile wallets), requiring changes to the checkout page UI to display these options.
- **Database Changes**: If the new API supports more detailed transaction data (e.g., customer location, device details), the database schema may need to be updated to store this additional information.

Scenario: E-commerce Website

Coordinated Changes Needed:

- **Backend Logic Changes**: such as new authentication methods or response structures.
- **Testing and Validation**
- **Documentation Update**: The internal and external documentation (e.g., API documentation, user guides) should be updated to reflect the new payment process.

Change propagation in SOA



- an Entity typically refers to a distinct unit that represents data or a business concept.

Consistency constraints

- To support change propagation, it's essential to ensure the system **maintains consistency**.
- We can use **consistency constraints** (defined using Object Constraint Language (OCL)) to **identify** inconsistencies.
- These constraints **act as rules** that the system must adhere to,
  - when violated, they could reflect the change has caused an inconsistency.

OCL Constraint

- Imagine a university enrollment system with two main entities:Student and Course
- The system enforces a rule that each student can enroll in a maximum of 5 courses per semester. This rule is defined as a **consistency constraint** using Object Constraint Language (OCL).
  - context Student
  - inv MaxCourses:
  - self.courses->size() <= 5

- **Initial Change:**
  - The university decides to add a new rule that allows **honor students** to enroll in up to 7 courses per semester instead of 5.

OCL Constraint

- If any of the constraints are violated during the enrollment process (e.g., a student tries to enroll in more than the allowed courses), the system will trigger an error, signaling that the change has introduced an **inconsistency**.

- To support this change and ensure the system remains consistent, we can:
  - Student Entity Update: A new attribute **isHonorStudent** must be added to the Student entity
  - Modify OCL Constraint:
    - if self.isHonorStudent then
    - self.courses->size() <= 7
    - else
    - self.courses->size() <= 5
    - endif

Propagating changes by fixing inconsistencies

- Ways of **resolving** a fact/rule violation (i.e. inconsistencies) are represented as Belief-Desire-Intention (BDI) plans, i.e. **repair plans**.
  - model multiple options
  - model the cascading nature of change propagation.

Repair plans

- Repair plan can be formally defined with the structure:
  - Triggering event: The **occurrence** that initiates the plan (e.g., an inconsistency).
  - Context condition: The **conditions** under which the plan can be applied.
  - Plan body: The **actions** taken to resolve the inconsistency.

Repair plans

- Plan = triggering-event : context-condition <- plan-body

1. Event e1 occurs
2. Plans 1-3 relevant to handle e1
3. Assume c1 & c3 true, c2 false
4. Plans 1 & 3 are applicable
5. Select (e.g.) plan P1 and start executing b1
   - b1 which may include primitive
     repair actions (e.g. add, create, modify, etc.) and/or sub-events, hence cascade.

Plan library

Plan1 = e1 : c1 <- b1

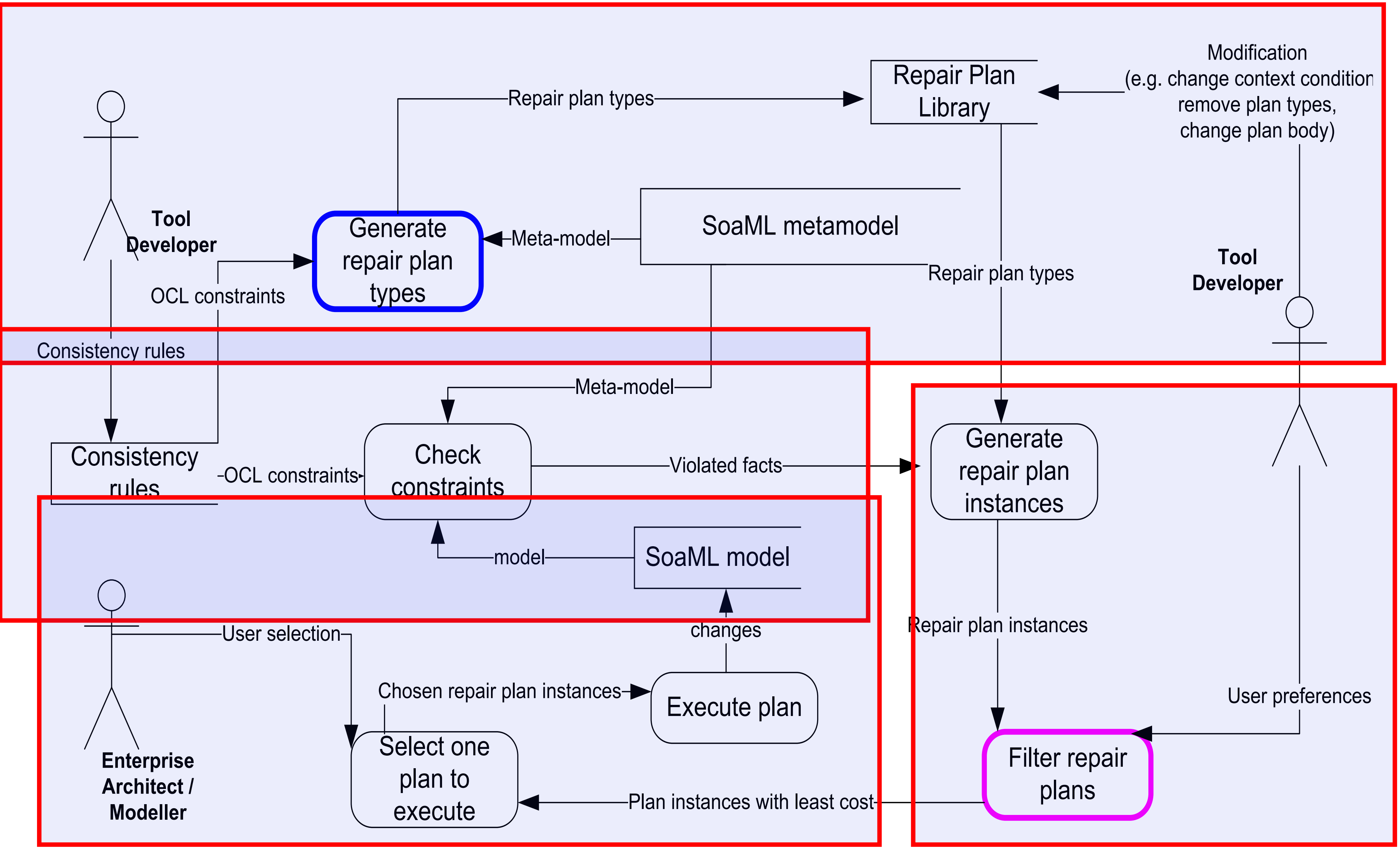
Plan2 = e1 : c2 <- b2

Plan3 = e1 : c3 <- b3

Plan4 = e2 : c4 <- b4

....

Change propagation framework for SoaML



Change propagation framework for SoaML

- At the **design stage**, a repair administrator defines consistency constraints using OCL.
- The **repair plan generator** uses the OCL constraints as inputs, and produces a set of event-triggered repair plans
  - form a library of solutions.
  - used by the change propagation engine to resolve constraint violations.
- **Repair plans are generated ahead of time**, but:
  - at runtime, the design is checked against OCL constraints.
  - If a violation occurs, a repair plan is selected and executed to resolve the issue.

Change propagation framework
Repair/change option selection

- Problem: how to select between different applicable (repair) plan instances to fix a given constraint violation?

- **Option 1:** Calculating **the cost of each repair plan** instance based on a set of primitive costs, and choosing the cheapest plan.
  - The cheapest cost heuristic may not always lead to the best way to resolve inconsistencies
  - Choice amongst alternative repair plans are necessarily **driven by domain specific consideration**, and cannot be adequately captured in a cost-based approach.

- **Option 2**:
  - the best inconsistency resolution is the one for which the resulting model, after having fixed all violations, is "**conceptually closest**" to the original model.
  - adopting a **minimal-change approach** to filter repair options in our change propagation framework
  - focus on **service choreography** in a SoaML model
    - focuses on the sequence and rules of interactions between multiple services

Change propagation framework
Select repair plans

- Encode this representation of a service choreography (i.e. UML activity diagram) into semantically-annotated diagrams called **Semantic Process Networks (SPNet)**

- A SPNet is a digraph **<V, E>** in which each **node** is of the form **<ID, nodetype, owner>** and each **edge** is of the form **<<u, v>, edgetype, condition>.**
  - Each event, activity, decision, or fork/join in an activity diagram maps to a node.
  - The **owner** attribute of a node refers to the service role

- Based on the SPNets, we then define a class of proximity (similarity) relations that allow us to compare alternative modifications of a **service choreography** in terms of how much they deviate from the original model.
  - Semantic proximity
  - **Structural proximity**

Structural proximity

- Each SPNet is associated with a proximity relation
  - $spn_i \leq_{spn} spn_j$: **spn$_i$** is closer to **spn** than **spn$_j$**
  - The proximity relations can be defined in a number of ways to reflect various intuition, e.g. set cardinality-oriented proximity measurement.

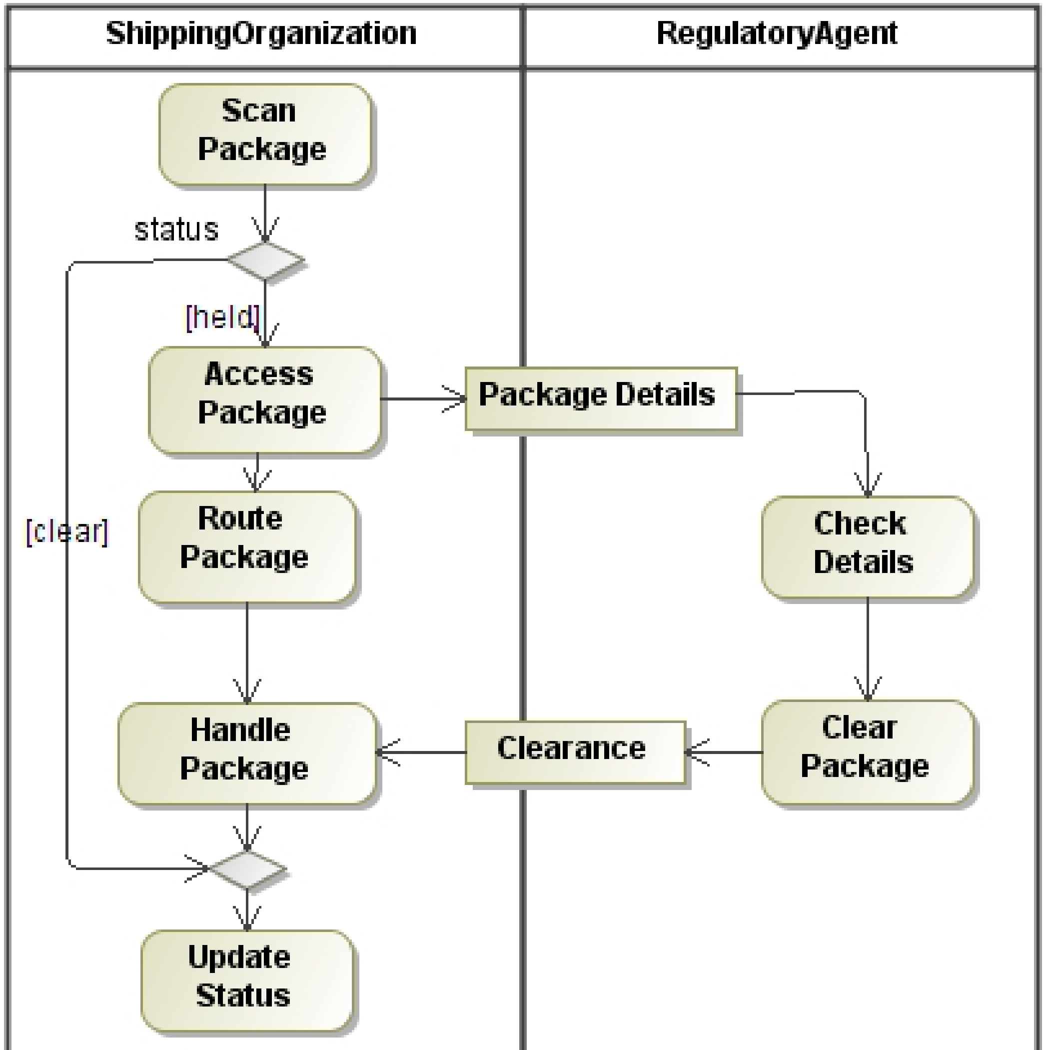
$$spn_i \leq_{spn}^{E} spn_j \text{ iff } |E_{spn} \triangle E_{spni}| \leq |E_{spn} \triangle E_{spnj}|$$

$|A|$ denotes the cardinality of set A

A $\triangle$ B denotes the symmetric difference of sets A and B

- The **symmetric difference** of two sets A and B is a set that contains elements which are in either A or B, **but not in both**.

Structural proximity
Example

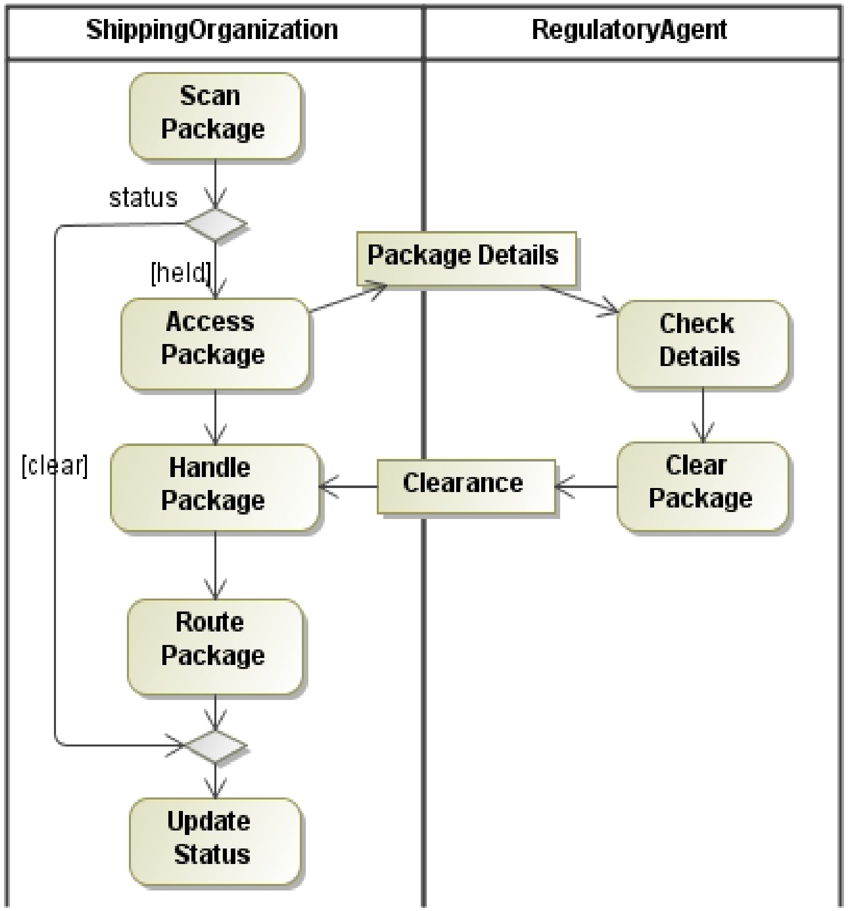


Original “Regulating Service” **service choreography (SC0)**

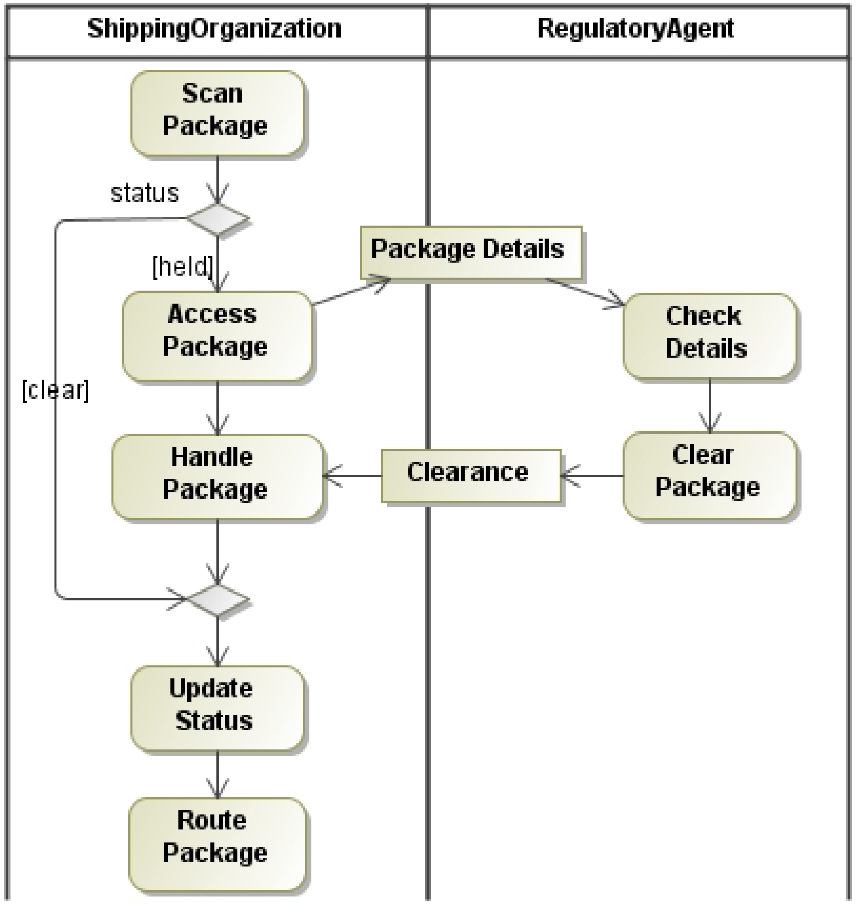
Domain-specific constraint:
“*Packages known to be held by a regulatory agent must not be routed by a shipping organization until the package is known to be cleared by the regulatory agent.*”

This domain-specific constraint is violated in the original service choreography.

Structural proximity Example (cont.)



Resolved "Regulating Service" service choreography (SC1)



Resolved "Regulating Service" service choreography (SC2)

Structural proximity Example (cont.)

- Now we need to select between SC1 and SC2.
  - Convert SC0, SC1 and SC2 into its SPNet representation
  - Calculate the edge difference between SC0 and SC1, and between SC0 and SC2
  - Select the one that “closer” to SC0

Structural proximity Example (cont.)

- 1->0:
  - AssessPackage->HandlePackage (SC1)
  - HandlePackage-> RoutePackage (SC1)
  - RoutePackage-> DecisionNode (SC1)
  - AssessPackage->RoutePackage(SC0)
  - RoutePackage->HandlePackage (SC0)
  - HandlePackage-> DecisionNode (SC0)
- 2->0:
  - AssessPackage->HandlePackage (SC2)
  - UpdateStatus->RoutePackage (SC2)
  - AssessPackage->RoutePackage (SC0)
  - RoutePackage->HandlePackage (SC0)

It means that SC_2 is closer to SC_0 than SC_1 and consequently SC_2 is the preferable repair/change option