Lecture 5: Outline

- Reminder: functions
- Varieties of functions
- Service-oriented systems
- Example: a master component of a service-oriented system
- Homework: augmented hotel system

Functions (reminder)

- Functions form a special class of relations that satisfy additional requirement: any element of the source set can be related to no more than 1 element of the target
- Functionality requirement mathematically:

$$\forall x, y, z.\ \ (x \mapsto y) \in R \ \wedge \ (x \mapsto z) \in R \ \Rightarrow \ \ y = z$$

- Any operation applicable to a relation or a set is also applicable to a function. For example, we can talk about the domain and the range of a function or a function as a set of pairs
- If f is a function, then $f(x)$ is the result of the function f for the argument x

Total and partial functions (reminder)

- Functions are called *total* if their domain is the whole source set
  Syntax: $f \in S \rightarrow T$ (or ascii $f : S$ --> T)
  where $dom(f) = S$ and $ran(f) \subseteq T$

- Functions are called *partial* if their domain is a subset of the source set
  Syntax: $f \in S \nrightarrow T$ (or ascii $f : S$ +-> T)
  where $dom(f) \subseteq S$ and $ran(f) \subseteq T$

Functional/array assignment (reminder)

- The notation used to describe machine actions (i.e., assignments in the machine events) allows us to directly assign values to indexed elements of arrays:

$$a(i) \ := \ E$$

- This is just syntactic sugaring for the following assignment:

$$a \ := \ a \lhd\!\!\!- \ \{(i \mapsto E)\}$$

- The assignment also works if a is modelled as a partial function. However, if we want to check/read values from such an array, we have to ensure/prove (by using the event guards and/or machine invariants) that the used index belongs to the function domain, i.e., $i \in dom(a)$

- $reserved(r) \ := \ FALSE$ **OK!**
- $nItems(j) \ := \ nItems(i) + 1$ **only if** $i \in dom(nItems)$

Varieties of functions

Suppose we have a function f (from the source X to the target Y). Then it is called

Total function	$\rightarrow$	-->	if $dom(f) = X, \ ran(f) \subseteq Y$
Partial function	$\pfun$	+->	if $dom(f) \subseteq X, \ ran(f) \subseteq Y$
Total injection	$\rightarrowtail$	>->	if $dom(f) = X, \ ran(f) \subseteq Y$ and one-to-one function
Partial injection	$\pinj$	>+>	if $dom(f) \subseteq X, \ ran(f) \subseteq Y$ and one-to-one function
Total surjection	$\twoheadrightarrow$	->>	if $dom(f) = X, \ ran(f) = Y$
Partial surjection	$\psurj$	+->>	if $dom(f) \subseteq X, \ ran(f) = Y$
(Total) Bijection	$\bij$	>->>	if $dom(f) = X, \ ran(f) = Y$ and one-to-one function

Varieties of functions: examples

- Injection: a function with 1-to-1 relationship between the source and target sets (e.g., an array without repeating elements)
- Example: $VU\_id \in PERSON \rightarrowtail\!\!\!\!\!\!\!+ \; ID$

  It is a partial injection: not all persons have a VU identification number, however, id is unique for each person
- Advantage: a reverse relation for an injection is also a function!
- Example: $VU\_id^{\sim} \in ID \rightarrowtail PERSON$

  A total injection this time
- Other examples:
  $Capital \in$ COUNTRY $\rightarrowtail CITY$, and
  $Capital\_of \in CITY \rightarrowtail\!\!\!\!\!\!\!+ \; COUNTRY$ (where $Capital\_of = Capital^{\sim}$)

Varieties of functions: examples

- Surjection: a function that covers all the target set
- Example:
  $married \in WIFE \twoheadrightarrow HUSBAND$

  It is a total surjection
- Another example:
  $Capital\_of \in CITY \mathrel{+\!\!\!\twoheadrightarrow} COUNTRY$

  A partial surjection this time

Varieties of functions: examples

- Bijection: a total function that is both injection and surjection
- Example:
  $married \in WIFE \rightarrowtail\!\!\!\!\twoheadrightarrow HUSBAND$

  It is a bijection (in many countries)
- Bijections relate sets with the same power (length)
- Another example:
  $VU\_account \in ID \rightarrowtail\!\!\!\!\twoheadrightarrow VU\_ACCOUNT$

  A bijection: both sets are of the same length and one-to-one relationships in both directions

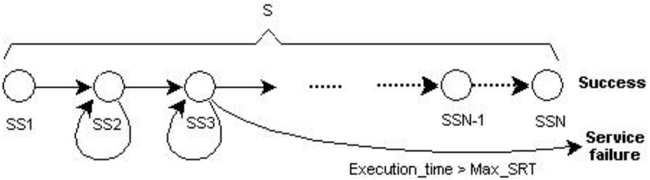
Telecommunicating and service-oriented systems

- Examples: telecommunication networks, service-oriented architectures, web services, cloud-based services
- Usually consists of multiple components that collectively provide a service to the service consumer (the external user or possibly other service provider)
- The components can be further partitioned into those that are responsible for service orchestration (management) or service execution
- Components for service orchestration: master components, service directors, service managers, (sometimes) frontend components
- Components for service execution: worker components, standalone components

Service decomposition

Often, a service request is decomposed and forwarded to different components (sub-service providers)

Service directors / master components are responsible for managing and controlling the whole service flow, forwarding requests to the respective components providing the required sub-services



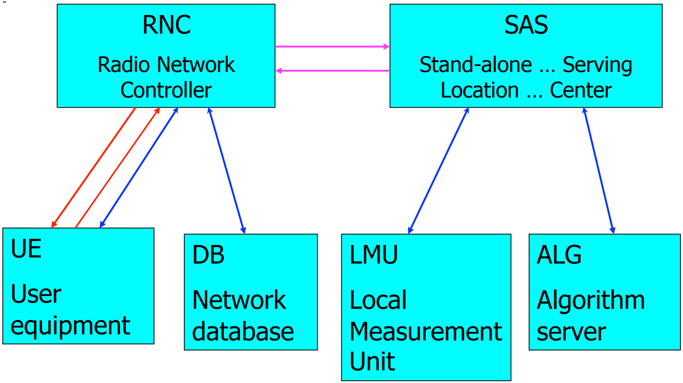
Telecommunicating and service-oriented systems (cont.)

- Components can be unavailable/busy or fail during execution
- A master component is responsible for monitoring the availability and failure status of its workers
- A master component can also incorporate fault tolerance mechanisms (e.g., retrying a service request, finding a replacement worker component, etc.)
- A master component can usually handle multiple service requests at once, while worker components are often executing a single service request

Positioning service (Nokia)

Typically, service-oriented systems have a layered architecture

Here, RNC (Radio Network Controller) – the external master (frontend) component, SAS – local service master component, while UE, DB, LMU, and ALG are employed as standalone / worker components



Simple example of a service-oriented system: requirements

1. The system (master component) handles the incoming service requests, distributing them to the available worker components
2. While being handled, the arrived service requests are stored in the input buffer
3. The input buffer has the pre-defined size that cannot be exceeded
4. Once handled, all the requests are stored in the output buffer (no size restrictions)
5. Each request has the handling status, which can be Waiting, Executing, Completed, or Failed
6. If the input buffer is full, a request is immediately transferred to the output buffer with the status Failed

Service-oriented system: requirements (cont.)

7. After an arrived request is added to the input buffer, it gets the status Waiting
8. Once a request from the input buffer is assigned to one of the available worker components, it gets the status Executing
9. Once a request is successfully handled by a worker component, it gets the status Completed and is transferred from the input buffer to the output buffer
10. A worker can fail to handle a request. Then the request gets the status Failed and is transferred from the input buffer to the output buffer

Service-oriented system: requirements (cont.)

11. Each request is associated with a specific service type
12. Each worker component is capable to handle a specific subset of service types. This knowledge is known beforehand and does not change during system execution
13. There is a number of active worker components at a particular moment. Any inactive worker can be activated and any active worker (not handling any requests) can be deactivated
14. Only an active worker component that is capable to handle the service type of a request can be assigned that request
15. A worker component can handle no more than one request at the moment

Service-oriented system: context

```
CONTEXT
   Services_ctx
SETS
   REQUEST
   RTYPE
   WORKER
   STATUS
CONSTANTS
   Waiting, Executing, Completed, Failed,
   Req_type, Worker_types, buffer_size
...
END
```

AXIOMS

$partition(STATUS, \{Waiting\}, \{Executing\},$
$\{Completed\}, \{Failed\})$

$Req\_type \in REQUEST \twoheadrightarrow RTYPE$

$Worker\_types \in WORKER \rightarrow \mathbb{P}_1(RTYPE)$

$buffer\_size \in \mathbb{N}_1$

END

Revealing a structure of abstract type elements

- Abstract types (like *REQUEST* or *WORKER*) are convenient to introduce standard notions of the modelled system
- If we need to reveal the internal structure of abstract type elements, we can introduce constant functions (like *Req_type*) in the context component to extract the necessary information from abstract type element
- Such functions are similar to a way object attributes (fields) are accessed in OOP
- Another possible function:
  $Req\_priority \in REQUEST \rightarrow PRIORITY$

Service-oriented system: machine

MACHINE

Services_mch

SEES

Services_ctx

VARIABLES

input, output, rstatus, active, assigned

INVARIANT

$input \in \mathbb{P}(REQUEST)$

$output \in \mathbb{P}(REQUEST)$

$rstatus \in REQUEST \nrightarrow STATUS$

$active \in \mathbb{P}(WORKER)$

$dom(rstatus) = input \cup output$

$input \cap output = \varnothing$

$card(input) \leq buffer\_size$

...

Service-oriented system: machine (cont.)

$$assigned \in active \leftrightarrow REQUEST$$
$$ran(assigned) \subseteq input$$
$$\forall r \cdot r \in ran(assigned) \Rightarrow rstatus(r) = Executing$$
$$\forall r \cdot r \in output \Rightarrow rstatus(r) \in \{Completed, Failed\}$$
$$\forall r \cdot r \in input \Rightarrow rstatus(r) \in \{Waiting,\ Executing\}$$

INITIALISATION

$$input,\ output\ :=\ \varnothing,\ \varnothing$$
$$rstatus,\ assigned\ := \varnothing,\ \varnothing$$
$$active :\in \mathbb{P}(WORKERS)$$

...

Service-oriented system: machine (cont.)

EVENTS

$request\_arrival \;=\; \textbf{ANY}\; r$
  $\textbf{WHERE}\; r \notin dom(rstatus) \wedge card(input) \leq buffer\_size$
  $\textbf{THEN}$
    $input \;:=\; input \cup \{r\}$
    $rstatus(r) \;:=\; Waiting$
  $\textbf{END}$

$request\_rejection \;=\; \textbf{ANY}\; r$
  $\textbf{WHERE}\; r \notin dom(rstatus) \wedge card(input) = buffer\_size$
  $\textbf{THEN}$
    $output \;:=\; output \cup \{r\}$
    $rstatus(r) \;:=\; Failed$
  $\textbf{END}$

...

Service-oriented system: machine (cont.)

...

$request\_assignment \;=\; \textbf{ANY } r, w$

$\quad \textbf{WHERE } r \in input \wedge w \in active \wedge rstatus(r) = Waiting$

$\qquad\qquad Req\_type(r) \in Worker\_types(w)$

$\quad \textbf{THEN}$

$\qquad assigned(w) \;:=\; r$

$\qquad rstatus(r) \;:=\; Executing$

$\quad \textbf{END}$

$request\_completed \;=\; \textbf{ANY } r$

$\quad \textbf{WHERE } r \in ran(assigned)$

$\quad \textbf{THEN}$

$\qquad output := output \cup \{r\}$

$\qquad input := input \setminus \{r\}$

$\qquad assigned := assigned \rhd \{r\}$

$\qquad rstatus(r) \;:=\; Completed$

$\quad \textbf{END}$

Service-oriented system: machine (cont.)

$\ldots$

$$
\begin{array}{l}
\quad request\_failed \;=\; \textsf{ANY}\ r \\
\qquad \textsf{WHERE}\ r \in ran(assigned) \\
\qquad \textsf{THEN} \\
\qquad\quad output := output \cup \{r\} \\
\qquad\quad input := input \setminus \{r\} \\
\qquad\quad assigned := assigned \rhd \{r\} \\
\qquad\quad rstatus(r) \;:=\; Failed \\
\qquad \textsf{END} \\
\quad worker\_activate \;=\; \textsf{ANY}\ w \\
\qquad \textsf{WHERE}\ w \notin active \\
\qquad \textsf{THEN}\ active := active \cup \{w\}\ \textsf{END} \\
\quad worker\_deactivate \;=\; \textsf{ANY}\ w \\
\qquad \textsf{WHERE}\ w \in active \wedge w \notin dom(assigned) \\
\qquad \textsf{THEN}\ active := active \setminus \{w\}\ \textsf{END}
\end{array}
$$

Invariants: summary

Different kinds/forms of invariants:

1. Typing invariants:
   $x \in S$, $y \subseteq S$, $r \in S \leftrightarrow T$, $f \in S \nrightarrow T$, ...
2. Interrelationships between different variables (and constants) :
   $input \cap output = \varnothing$, $f \subseteq r$, $dom(f) = y$
   $card(input) \leq buffer\_size$

   The variable values can be used to directly define the typing of other variables:
   $items \in POW(ITEMS)$, $price \in items \rightarrow NAT$

Invariants: summary (cont.)

3. Conditional (local) invariants of the form
   $cond_1 \ \wedge ... \wedge \ cond_k \ \Rightarrow \ property$

   $served = TRUE \ \Rightarrow \ payed = TRUE$
   $choice \neq None \ \wedge \ served = FALSE \ \Rightarrow \ nItems(choice) > 0$

   In such a way, invariants can be "narrowed" down to specific points of system execution

4. Using quantifiers in invariants on more complex data structures:
   $\forall r \cdot r \in ran(assigned) \Rightarrow rstatus(r) = Executing$
   $\forall r \cdot r \in output \Rightarrow rstatus(r) \in \{Completed, Failed\}$

Homework: an extended hotel booking system (slightly changed requirements from the first homework)

1. The hotel booking system handles room reservation by customers;
2. The system must have operations (events) for room reservation, cancellation, customer check-in (with a reservation), customer check-in (without a reservation), and customer check-out;
3. Each reservation is stored by the system until it is cancelled or the reservation customer checks-in (with a reservation);
4. A reservation stores the information about the reserved room, the reserved dates, and the customer;
5. For any reservation, its dates cannot overlap with any other reservation dates for the same room;

Homework: an extended hotel booking system (requirements, cont.)

6. A reservation can be cancelled;
7. After cancellation, the stored reservation is removed from the system;
8. After a customer's check-in (with a reservation), the reserved room gets the status "occupied" and the stored reservation is removed from the system;
9. The system keeps the information about the occupied rooms, the customers, and the dates they are staying;
10. After a customer's check-in (with a reservation), the information from the room reservation is associated with (copied to) that of the occupied room;

Homework: an extended hotel booking system (requirements, cont.)

11. For any occupied room, its dates cannot overlap with any reservation dates for the same room;
12. A customer can check-in (without a reservation), if there is an unoccupied and unreserved room for the specified dates;
13. Once a customer checks-out, the information about the occupied room (the customer and dates) is removed from the system.

Homework: a hotel booking system (cont.)

Hints:

- Again, carefully define the necessary data structures in the model context
- Hotel reservations can be introduced as an abstract set, while the corresponding constant functions can be defined to extract necessary information (like the customer or dates) from them
- A similar or the same approach can be applied to model the info about the occupied rooms
- Dates can be also modelled as elements (subsets) of a pre-defined abstract set. Alternatively, a subset of natural numbers can be used
- The requirement 5 and 11 must become model invariants