Overview

We want to connect web architecture to state machines. In the document, we come from the direction of web architecture, not from the direction of state machines (as in the paper).

Web Architecture

The web is a distributed system with resources as core abstraction and transfer of resource representations via request/response interactions between client and server. Resources can be created and deleted, and resource representations can be read and written. In other words, web architecture supports a CRUD (create-read-update-delete) interface abstraction.

Resource representations can contain links and forms, also known as affordances. With affordances, the server tells the client about possible interactions on resources. The client chooses from the server-provided affordances, following links or filling in forms to progress towards a goal.

Resources can evolve over time via unsafe requests, i.e., requests that change the state of the resource.

Process, Occurrence and Interaction

Evolution of resource state is described via processes.

Processes are encoded in state machines (via transition table) and describe how states should progress on all occurrences of the process via interactions, i.e., request/response pairs.

Occurrences capture the runtime perspective. Occurrences link to processes and have an actual state. Occurrence representations give the "ego" view on how states could progress via the next possible interactions/affordances (and who says that links and forms show only the next interaction (and state) and not the next n interactions and states?).

Interactions have a name and consist of a request/response pair. Interactions can be "ground" or contain variables via template URIs (RFC 6570 URI Template). Interactions form the basis for the transition tables representing the progress of state.

Components and Fragments

Components and fragments describe partial processes. Components have an inital state. Fragments do not have an initial state.

Components and fragments are per resource (a single "port"). Atomic processes are per

resource (a single "port"), while composite processes may describe the behaviour of mulitple resources (multiple "ports").

Execution (or Simulation?)

Processes can be executed on the origin server (having a server connector) and on the user agent (having a client connector).

Each occurrence could track its evolution, i.e., have a local trace (a record of its history). Both client and server connectors can record the interactions, i.e., request/response pairs, from their point of view. So we arrive at client traces and server traces.

A proxy could track all interactions between all client and server connectors ("God" view on ground interaction level). So we arrive at an interface trace, which can be represented as sequence diagram.

Composition

We need ways to compose multiple processes, e.g., via parallel composition of processes P1 and P2, written as P1 || P2 or serial composition, written as P1 ○ P2.

Compositions can give rise to sequence diagrams that show the interactions between processes (or occurrences?).

Properties

One could detect following properties on processes (or transition systems?):

- determinism
- reactivity
- ground vs. variables/templates
- finite trace vs. infinite trace

We could also list the interactions of a process or all "ports", i.e., all URIs to resources associated with a composite process.

Create-Read-Update-Delete Operations

With web architecture, we have CRUD. The operations can be split into two groups: first, create and delete relating to the existence of resources, and second, read and update relating to the state of resources.

- Create: POST to a collection resource or PUT to a non-collection resource (man, the non-* resource terminology sucks). For creating a resource, POST to a collection resource yields a URI assigned by the server. For creating a resource via PUT, in principle the client could decide on a URI. However, in our case, the server returns a

link to a URI and then the user agent uses that URI as target URI for the PUT request.

- Delete: DELETE to any resource. Easy, but side effects must be handled.
- Read: GET to any resource. Read should return the state of the target resource of the request, plus possible affordances.
- Update: PUT to a non-collection resource. The representations of all resources are not always (completely) writeable. The state of resource A may influence the state of resource B. Also, the server adds the affordances of occurrences based on the current state, so we do not want to overwrite the affordances.

Unfortunately, the creation and deletion of objects is not something common in the state machine literature. Or the creation and deletion of state machine occurrences. But object creation and deletion is key in web architecture. SDL from the ITU has a create action, though, which can create new agents (and as part of a new agent presumably a new occurrence of a process).

We do not have:

- Remote-procedure call: POST to a non-collection resource (e.g., for resetting a state machine).

For that, we would need some sort of language to describe the procedure to be called.