Model-based Orchestration

Last updated: 13th February 2008

For comments: andrew.farrell@hp.com

Background

The purpose of this work is to provide a model-based approach to component orchestration, facilitating natural modelling with support for verification of properties pertaining to the integrity of orchestrations and for enactment.

This document is structured as follows...

((The document will be largely unstructured whilst the approach is formative))

Key Notions

- The components to be orchestrated will be possibly distributed.
- We want to be agnostic with respect to transport mechanisms, but we will consider the use of Web 2.0 technologies such as RSS/Atom feeds, AtomPub, RESTful interfaces, ... We will also consider the case of the deployment context being SmartFrog which (for now) utilises RMI. Notably, the core implementation of the approach should be made to be independent of SmartFrog so that it can operate as a standalone technology.

  For the time being, then, we make no assumptions regarding transport, except that there is some unspecified means for it.

- The authoring of workflow-like artefacts should be supported. We also need to support the notion that components may have arbitrary life-cycles (or state machines) associated with them, and transitions within such machines may be arbitrarily predicated on the state of the rest of the orchestration model. This is because sometimes workflow-like artefacts are more useful, but other times lifecycle artefacts. As we shall note neither can be mapped to a subset of the other. Furthermore, the leaf activities of the workflow artefacts are going to be components of arbitrary life-cycles.

Orchestration Meta-model

An enumeration of the constructs of the meta-model defined for orchestration follows. We will explain them in due course.

- Component (with arbitrary life-cycle)
- Composite – scoping artefact
- Workflow artefacts
  - Synchronisation types – Go and Stop
  - Sequence and Unordered Sequence types –Seq, SeqCancel and UnorderedSeq
  - Parallel and Priority Parallel – Par and PriPar
  - Exclusive Choice with and without default – DefaultChoice and Choice
  - Deferred Choice – DeferredChoice
  - Trivial Completion – Empty
  - Free Choice – FreeChoice
  - Multiple Choice – MultiChoice
  - Multiple Merge – Multimerge
  - Discriminator m from n – Discriminator
  - Multiple-Instance Activities – Multi*
  - Cancel Activity – CancelActivity
  - Cancel Case –Exit
- ISA hierarchies

Component

A component defines a state machine which may be characterised as follows.

Component State Machine

A component state machine is a quintuple: (n, S, A, D, T), where:

- n – a unique name of the component
- t – a type name for the component
- S is a finite set of states of the component, ranged over by s, s', ...
- A is a finite set of action names, ranged over by a,a',...
- D is a finite set of dependency propositions, ranged over by d, d', ...
- T is a finite set of transitions. A transition is a product of a source state, a destination state, an action name, and a dependency proposition, viz. $T \subseteq S \times A \times D \times S$

Dependency

A dependency is an arbitrary boolean expression over the states of other components. The set of dependencies D_c that may be used in the definition of a state machine for component c is built up inductively and is the smallest set containing the following artefacts:

- Any pair (n, s) where n is the name of a component, c' (c'$\neq$c), within the same composite as c – see below, and $s \in S$, where S is the set of states of c'

- !e where e$\in$D (! means “not”)
- e + e’ where e, e’ $\in$D (+ means “or”)
- e | e’ where e, e’ $\in$D (+ means “and”)