A generic OSGi-based Model Framework for Delivery Context Properties and Events

Jaroslav Pullmann,[1] Yehya Mohamad,[1] Carlos A Velasco,[1] Stefan P. Carmien[2]

[1]Fraunhofer Institute for Applied Information Technology FIT, Schloss Birlinghoven, 53757 Sankt Augustin, Germany
[2]Tecnalia Research, Paseo Mikeletegi 7, Parque Tecnológico, E-20009 San Sebastián, Spain

{carlos.velasco, yehya.mohamad, jaroslav.pullmann}@fit.fraunhofer.de
stefan.carmien@tecnalia.com

Abstract. Content adaptation systems rely on standards-based modeling of user needs and preferences, rendering platforms, assistive technologies and other relevant aspects of the overall delivery context. Despite their differing domains, these models overlap largely in respect of their digital representation and handling. We present hereby our work on a generic model framework exhibiting a novel set of features developed to tackle commonly found requirements in the area of user and delivery context modeling.

Keywords: delivery context model, user preferences, content adaptation, OSGi, reactive systems

1 Introduction

The recurrent requirements for user, device and delivery context modeling has led to the development of a generic model framework based on the OSGi module runtime environment for Java. The OSGi platform gained a wide acceptance as flexible deployment model of dynamic modular applications (bundles) ranging from mobile appliances to enterprise deployment scenarios. Our framework comprises a set of OSGi bundles exposing services for model maintenance, querying and dynamic management of data sources (sensors) used to populate the models. The underlying model representation allows for storage of hierarchical, ordered, highly structured data independently of their underlying data-model – RDF triples or XML info sets.

2 Generalized Model Representation

2.1 Abstract Data-Model

There are various data structures commonly used to encode modeling vocabularies: XML trees (acyclic directed graphs), RDF triple graphs[1] in different serializations, JSON[2], simple property-value mappings (Java property files), etc. To relieve model engineers and users from considering the peculiarities of either data structure and to allow an uniform management and query interface, a *generalized, graph-based storage architecture* has been developed, capable of capturing hierarchical, ordered, highly structured data and adherent metadata. It considerably simplifies the import and aggregation of external profile sources.

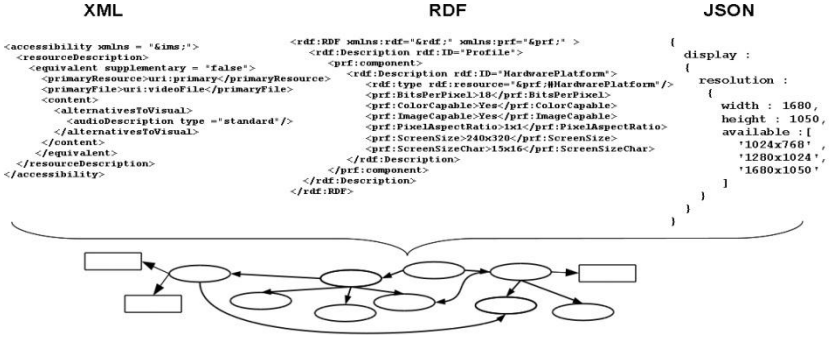

Fig. 1. Generalization of different input data structures.

Orthogonal to technical aspects, the model representation is considered generic in respect to data purpose, granularity and repository organization. Thus user preferences, location data, ambient contexts or home appliance properties are equally well captured by named model instances within the same framework supporting cross-referencing and joined queries. No organizational paradigm is enforced. In contrast to traditional single containment hierarchies (collections, directories) individual model instances are assignable to arbitrary number of functional groups that cover a particular shared "aspect":

- common data and metadata (group name, access rules, provenience, etc.) with various insertion semantics (merge, append, replace) and conflict resolution rules
- validity constraints, computed values and reactive rules via *model item properties*
- named queries mapped to virtual top-level nodes of a model

A subset of the XPath 2.0[3] language has been adopted to navigate and query the repository. Path expressions are the means of selecting nodes, defining model item pro-

1 RDF bindings for UAProf; Device Profile Evolution Architecture, etc.
2 http://www.json.org/
3 http://www.w3.org/TR/xpath20/

perties and linking them to individual models or groups. They leverage extension functions e.g. to retrieve model instances:

```
sys:model("pc")/display/resolution/width
```

The repository management internally employs and exposes a broader set of events. In addition to monitoring life-cycle stages (created, updated, deleted), value change and (in)validation of nodes, their usage context is considered as well. When external clients subscribe, repository events or input sources (sensors) are added, removed or send they heartbeat signals, and corresponding events are triggered. There are dedicated path functions to test the existence (a) or to access payload of events raised on a context node <path> (b). These functions are used in trigger conditions of rules (see below):

```
a) boolean on:value-changed(<path>)
   boolean on:subscription-added(<path>)
b) String get:subscription-added-by(<path>)
   <T> get:value-changed-from(<path>)
   <T>  get:value-changed-to(<path>)
   ValueChangeEvent  get:value-changed-event(<path>)
```

2.2 Model Item Properties

The validation and processing of model nodes has been extended in spirit of XForms model item properties[4]. They pertain to a node set of an individual data instance. By means of model groups we extended their applicability to a set of comparable model instances as well. Models inherit group's item property definitions and may additionally define local ones.
A schema-based validation is not enforced. It depends on the original data model (XML Schema, RDF Schema, JSON Schema[5]), whereas validity conditions might by specified in a generic way by *constraint expressions*. They further benefit from the use of functions, node's context and its recent value, which makes them appropriate to express assumptions on value's correctness and plausibility:

```
Node user = registry.getModel("userId29");
user.select("//age").addConstraint(". > 18")
```

Nodes of one or multiple model instances might be correlated or have assigned values by a *calculation expression*. These declaratively state a dependency between the computed node, its input nodes and scalar arguments, which is automatically maintained across model updates.
Finally, reactive behavior can be associated with nodes by means of *rule expressions*. This comprises model updates and invocation of internal methods or external services whenever rule's trigger condition becomes satisfied. In contrast to approaches like (Assad et al., 2006) restricted to pattern matching on node's value the whole range of event-test and retrieval functions is available for the definition of trigger conditions.

4 http://www.w3.org/TR/xforms/#model-xformsconstraints
5 http://tools.ietf.org/html/draft-zyp-json-schema-03

```
Node device = registry.getModel("deviceId6");
  model.addRule("abs(
  get:value-changed-event('/monitor/temperature')/from-
  get:value-changed-event('/monitor/temperature')/to)
  ge 20",// suspicious temperature
  notify("admin@mydomain.com")
  );
```

Since this approach operates on recent values only we plan to introduce support for recognition of their continuous change over a time frame by means of complex event processing[6].

3 Architecture of the Model Framework

The generic model repository is the core service provided by the framework. Thanks to the R-OSGi[7] platform extension it is transparently exposed to local and external clients. In addition to location transparency there are obvious benefits of using an OSGi runtime container for application development:

- emphasis of the service interface: depending on the deployment scenario and the capabilities of the underlying device the service might be backed by different implementations while retaining the same interface.
- modularity, dependency management and hot deployment: explicit statement of dependencies at code (package) and service level along with their automatic management through the Service Component Runtime (SCR) yield to a compositional approach of building scalable, layered applications.
- mature standard service interfaces and various open source implementations.

Figure 2 outlines the architecture elements and their inter-dependencies. Within the upper layer reside framework clients like content adaptation services leveraging a graphical user interface to retrieve and manage their profile data and various sorts of sensors. The data exchange services map between the internal graph representation and the exchange formats. The generic model service employs an event bus service for internal distribution and processing of repository events. The subscription service manages event subscriptions and routes events to their respective recipients.

6 http://en.wikipedia.org/wiki/Complex_event_processing

7 http://r-osgi.sourceforge.net/

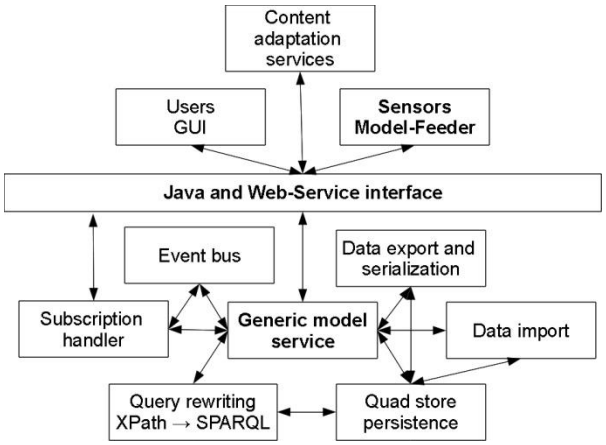

Fig. 2. Architecture components

The remaining sections focus on two practical aspects of model management that were of particular importance in our projects – implementation of context-aware input sources for (device) models and light-weight remote interfaces to model services.

3.1 Dynamic property sources

A novel aspect of recent technologies like DPE[8] and DCCI[9] is their explicit coverage of dynamic properties of the delivery context (Timmerer *et al*., 2010). Some examples are:

- system parameters: volume, display geometry and orientation
- system resources: memory, CPU usage, network bandwidth
- active processes of assistive technologies: screen readers, magnifiers
- attached hardware: Braille keyboard, USB devices

The actual sources of this information, their implementation, interaction with the host environment or considerations on quality of service parameters (sampling rate, overall reliability) seem to be out of the scope and are not discussed at all.

For the purpose of acquiring runtime parameters of host devices we leverage the OSGi runtime and a set of detection bundles. These consist of Java code, shell scripts, native libraries and other resources. Every bundle registers an implementation of the *ModelFeeder* interface which is expected to update model locations listed in the configuration property "exported-properties" by invoking methods of the *Repository* service. The component manifest may specify dependency on further *ModelFeeder*s providing a filter on their exported properties.

The OSGi Service Component Runtime takes care of enabling any component which mandatory dependencies are satisfied and injected via the named "bind"-method or disabling it, when these become unavailable. The code snippet illustrates a component

8 http://www.openmobilealliance.org/Technical/release_program/dpe_V1_0.aspx
9 http://www.w3.org/TR/DPF/

for detection of the open-source screen reader Orca, that states its dependency on another component supplying a list of active OS processes.

```
public interface ModelFeeder(){
    void setRepository(Repository r);
    void releaseRepository(Repository r);

    // life-cycle methods of this component
    void doActivate();
    void doDeactivate
    …
  }
<scr:component name="component-example"
  xmlns:scr="http://www.osgi.org/xmlns/scr/v1.1.0">
<implementation
class="com.imergo.modelservice.impl.readers.Orca"/>
<property name="model-id" value="localhost"/>
 <property name="exported-properties"
           value="device/sound/volume"/>
 <reference
  interface="com.imergo.modelservice.Repository"
  cardinality="1..1"
  bind="setRepository"
  unbind="releaseRepository/>
 <reference name="linux-helper"
  interface="com.imergo.modelservice.ModelFeeder"
  target="(&(platform.os=linux)(exported-
   properties=device/processes))"
  bind="setModelFeeder"
  unbind="releaseModelFeeder"
  cardinality="1..1"
  policy="dynamic"
  activate="doActivate"
  deactivate="doDeactivate"/>
 />
</scr:component>
```

3.2 Client interfaces to the model framework

When considering remote client interaction with the model framework following interface requirements were tackled:

- simplicity and loose coupling: this requirement has been satisfied by provision of a REST service interface. This allows for retrieval and filtering of models:

```
GET <server>/model/<id>
GET <server>/model/<id>/path/within/model
```

- scalability: the client should not be forced to retrieve data at the level of storage entities (model instance). Either a path-based query should be supplied or a named query stored on the server should be invoked in order to filter the result. Client queries relying on literal path expressions are brittle, since they require a close knowledge of the data model and might accidentally break when the model has structurally changed on the server side. To circumvent this dependency we introduced a named query interface. A query expression is stored on the server, associated with a model or model group and named using the CURIE notation. The client invokes this query as a virtual step within the associated model:

  ```
  GET <server>/models/<id>/my:epxr
  ```

- proactive (PUSH-oriented): the client should not be forced to maintain a local model cache in order to circumvent network round-trips. Instead it should be able to subscribe for particular event(s) notifications.

  ```
  POST <server>/models/<id>/path/within/model
  subscribe=get:value-changed-
  event&notify=http://myepr...
  ```

4 Conclusions

Our generic model framework significantly reduces the effort of managing delivery context models and their dynamic data sources. The data-structure agnostic storage and support for navigational and property-based queries has proven successful in deployment scenarios involving a multitude of modeling vocabularies and use cases. Further developments target at an extension by event and rule-based reasoning via additional OSGi Bundles in order to evaluate streams of user interaction and system events.

5 References

Assad, M.; Carmichael, D.J.; Kay J.; Kummerfeld, B. (2006). Active Models for Context-Aware Services. Technical Report 594. University of Sydney.

Timmerer, C.; Jaborning, J.; Hellwagner, H. (2010). A Survey on Delivery Context Description Formats – A Comparison and Mapping Model. Journal of Digital Information Management, 8 (1).