Enabling non-programmers to develop smart environment applications

Artem Katasonov

VTT Technical Research Center of Finland

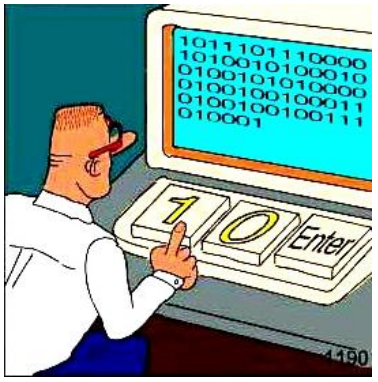
SISS at IEEE ISCC, Riccione, 22.06.2010

Long-term motivation

Most people know how to replace a light bulb (also to understand parameters, to acquire).

Constructing a home entertaining system from a TV, players, amplifiers, etc. does not take an electronics expert.

But when it comes to constructing or modifying software applications, **there exists no action small and easy enough** so that it would not require a person with a software engineering education.



Digital literacy

Although the human population becomes more and more fluent in interaction with software and the Web, **it's as if they can "read" but not "write**.

"Digital fluency" should mean *designing, creating, and remixing*, not just browsing, chatting, and interacting.

Resnick et al. (MIT) *Scratch: Programming for all*, CACM, Nov. 2009

Software in Smart Environments

Smart environments - small worlds where different kinds of smart device are continuously working to make inhabitants' lives more comfortable.

What is "more comfortable" is subjective and dynamic.

! A future, where smart environments are pervasive but software applications running in them can only be developed/modified by outsider software experts but not environments' inhabitants, **makes little sense**.

Example scenario

1. Jim just rented a car for a family holiday trip. The car has a *board computer with navigation* (BC) and a *Web-enabled passenger entertaining system* (ES).
2. Before starting the trip, Jim is presented on the screen of ES with a view to the software *currently running* in the smart environment of the car. Jim gets an understanding that: BC provides data on the location and destination, if set, of the car as well as the record of the renter; location data is automatically sent to the rental company for tracking; and ES is instructed to show automatically (when in certain mode) a weather forecast for the set destination of the car.
3. In few touches, Jim *modifies the existing program* so that weather is not shown automatically, but only when explicitly requested.
4. Jim then engages the *recommender function* ”what else can I do with the data available?”, and among options provided selects to display on ES a page from Wikipedia that is geo-tagged with a location closest to an input location, assuming the current location of the car in this case. He then instructs to engage this function automatically as car moves.

Example scenario (continued)

5. Jim triggers a *device discovery function* to connect his smartphone. The view on ES gets extended with smartphone's published services and components. Jim then instructs the phone to post to the smart environment data on the call status.
6. Jim browses a *list of generic tasks* defined for the smart environment, selects ”silence”, and then instructs to perform this task when his phone is ringing. A proper software component, to mute ES, is then automatically integrated into the application.
7. Jim browses the list of tasks again, selects ”display”, and maps its input to the record of the renter. He then attempts to specialize this task as display on the ES screen, but gets informed that displaying private data on the “public” display of ES *is not allowed due a company policy*. Therefore, Jim selects to display the record on his smartphone.
8. Jim presses ”done”. The application parts for ES and the smartphone are built, uploaded and deployed to the devices and start execution.

Target scenario properties

- High level of abstraction understandable for non-programmers.
- On-the-fly development.
- Flexibility with respect to adding new devices and software components.
- Combination of *task-based* (user knows what he wants) and *opportunistic* (possible actions are recommended based on available data) design.
- Ability to define policies to restrict users from designing unsafe applications.

Enabling non-programmer development

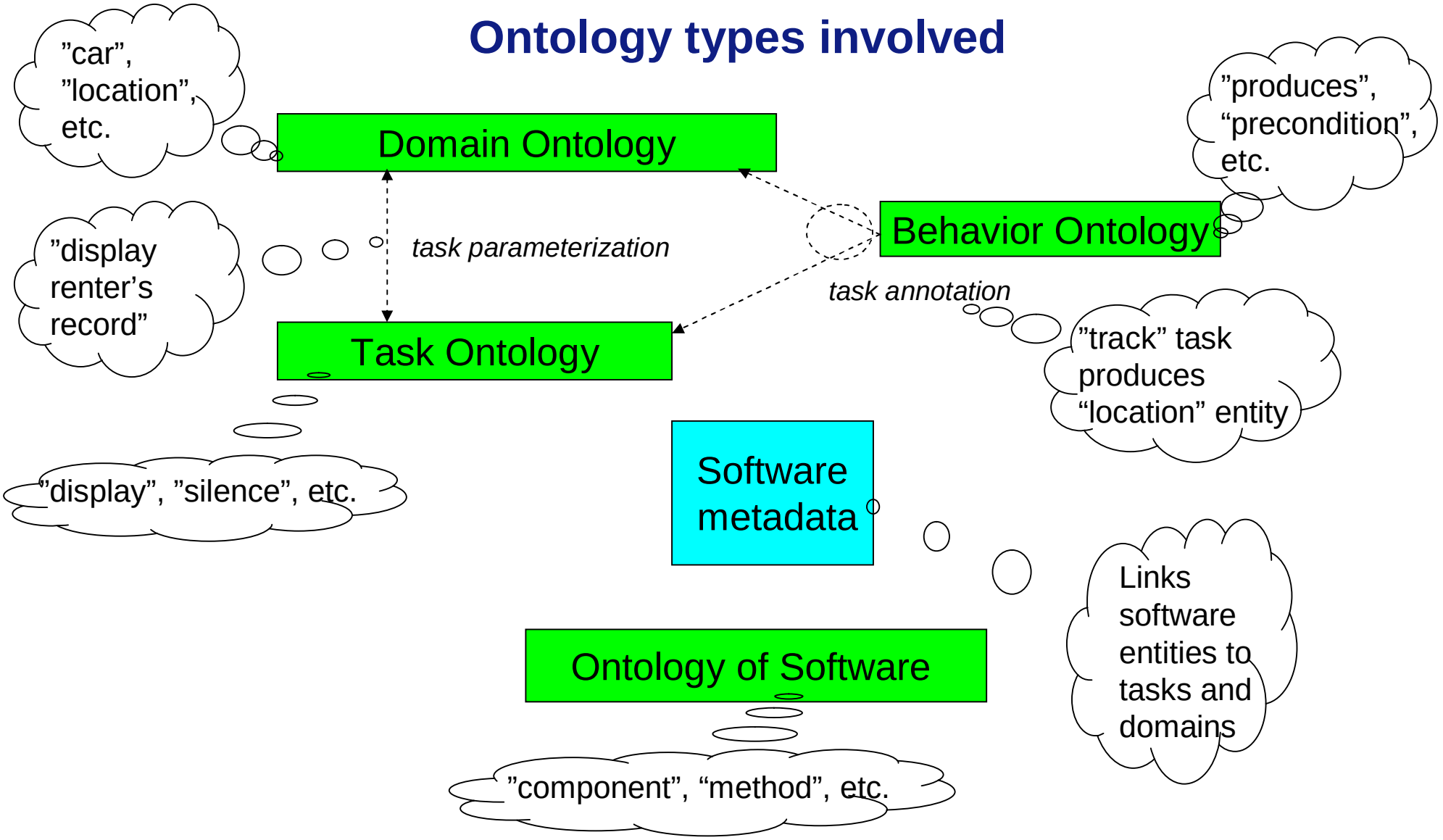
The general process is that of **Model-driven engineering**: the application is designed in a graphical modeling tool while the executable code is generated.

The **component-based approach** should ideally create a situation where the code of a new application is generated in full, i.e. it consists only of pre-existing components plus the code generated from the model.

Semantic component metadata has then to enable fully automatic transition between the platform-independent (tasks, data) and the platform-specific (particular software components) views.

Finally, **other applications of ontologies** have to further automate development of the model, in ideal case completely removing the need for the user to work with the model directly.

Ontology types involved



SOFIA Smart Modeller

Smart Modeller is Eclipse-based tool. It consists of a graphical editor and a set of plug-ins.

Plug-ins available at the moment:

- *Java Code Generator* – generates executable Java program from the model.
- *Python Code Generator* – generates executable Python program from the model.
- *Repository Exporter* – exports a part of the model into an RDF repository for later reuse.
- *Repository Importer* – imports model elements from a repository into the model.
- *SIB Subscription Generator* – generates SOFIA-specific data access elements based on a domain ontology.
- *Task Importer* – imports a task from a task ontology.
- *Implementation Finder* – finds from a repository a software implementation for a task, based on metadata.
- *Opportunistic Recommender* – suggests possible additions to the model, based on metadata.
- *Java Action Template Creator* – generates a stub for a Java action component when manual programming is needed (not for non-programmers).

1. Encoding application models with RDF

_:node1275642032940_2 rdf:type **model:Graph**; model:name "Action: Find Wiki Page";
rdfs:comment "Finds a Wikipedia page that is the nearest to a location";
model:contains _:node1275642032940_2_A, _:node1275642032940_2_R, _:node1275642032940_2_RC,
_:node1275642032940_2_P1, _:node1275642032940_2_P1C.
_:node1275642032940_2_A rdf:type **model:Action**; model:name "findByLocation"; model:implementation
"sofia.actions.dbpedia.PageFinder.findByLocation"; diagram:x "100"; diagram:y "100".
_:node1275642032940_2_R rdf:type **model:Parameter**; model:name "model:return"; model:position "1";
model:value ""; model:type "java.lang.String"; diagram:x "200"; diagram:y "100".
_:node1275642032940_2_RC rdf:type **model:Connector**; model:relationship "model:produces";
model:source _:node1275642032940_2_A; model:target _:node1275642032940_2_R.
_:node1275642032940_2_P1 rdf:type **model:Parameter**; model:name "wgs84Location"; model:position "1";
model:value ""; model:type "java.lang.String"; diagram:x "0"; diagram:y "100".
_:node1275642032940_2_P1C rdf:type **model:Connector**; model:relationship "model:has";
model:source _:node1275642032940_2_A; model:target _:node1275642032940_2_P1.

- This notation can be at any time produced for the edited model.
- Smart Modeller's repositories also use this notation.
- We implemented a software tool for automatic generation of such description for the methods of Java classes.

2. Defining the hierarchy of tasks

Part of “standard” task ontology:

task:Task rdf:type rdfs:Class.

task:TaskWithInput rdfs:subClassOf task:Task.

task:TaskWithOutput rdfs:subClassOf task:Task.

Example application task ontology:

ex:Display rdfs:subClassOf task:TaskWithInput.

ex:DisplayOnES rdfs:subClassOf ex:Display.

ex:AccessData rdfs:subClassOf task:TaskWithInput, task:TaskWithOutput.

ex:AccessRenterRecord rdfs:subClassOf task:AccessData.

ex:DisplayRenterRecordOnES rdfs:subClassOf ex:DisplayOnES;
task:includes ex:AccessRenterRecord.

3. Linking tasks to model elements via SPARQL

```
task:Task rdf:type rdfs:Class; task:modelPattern
    "?graph model:contains ?action. ?action rdf:type model:Action; model:name ?name".

task:TaskWithInput rdfs:subClassOf task:Task; task:modelPattern
    "?graph model:contains ?input. ?input rdf:type model:Parameter; model:position \"1\".
     ?connI rdf:type model:Connector; model:relationship \"model:has\";
            model:source ?action; model:target ?input".

ex:findByLocation rdfs:subClassOf task:TaskWithOutput , task:TaskWithInput ;
    rdfs:comment "Finds a Wikipedia page that is the nearest to a location" ;
    task:modelPattern
            "?action model:implementation \"sofia.actions.dbpedia.PageFinder.findByLocation\".
             ?input model:type \"java.lang.String\"" .
```

- The full pattern of a task is created as concatenation of model:pattern of all its super-classes and its own.

4. Annotating task parameters

ex:findByLocation
bhv:requires [model:type ct:GeoLocation; model:maps "?input"];
bhv:produces [model:type ct:WebLocation; model:maps "?output"].

ex:openPage
bhv:requires [model:type ct:WebLocation; model:maps "?input"].

- bhv:requires specifies a consumed data entity
- bhv:provides specifies a produced data entity
- model:maps in both cases specifies the input/output parameter – as a variable form from the task's SPARQL pattern

```
st:SubscribeClass rdfs:subClassOf st:Subscribe; task:modelPattern
                        "?input3 model:value \"http://www.w3.org/1999/02/22-rdf-syntax-ns#type\".
                         ?input4 model:value ?class.
                         ?connR rdf:type model:Connector; model:relationship \"model:produces\";
                                 model:source ?action; model:target ?output.
                        ?output model:name \"s\"";
    bhv:produces [model:type "?class"; model:maps "?output"].
```

4. Using other behavioral properties

ex:VoiceInform bhv:precondition
”?x rdf:type ex:ES; ex:volumeLevel ?vol. FILTER (?vol < 30)”.

ex:SilenceES bhv:effect ”?device ex:volumeLevel 0”.

5. Defining policies

ex:Renter policy:prohibited ex:DisplayRenterRecordOnES

ex:Renter policy:allowed ex:DisplayRenterRecordOnMobile