HOBO: A Hybrid Modelling Framework

Colin Puleston, Bijan Parsia

School of Computer Science, University of Manchester, United Kingdom

Abstract. HOBO is a Java hybrid modelling framework for building ontology-driven applications. A dynamic frame-based model provides a seamless integration of entities derived from (a) a Java object model, and (b) one or more externally-represented ontologies. Each entity retains dynamic links to its source(s), enabling the utilisation of both ontological reasoning and procedural processing in the dynamic updating of model instantiations. Although HOBO is totally independent of any particular ontology format, it was designed largely with OWL ontologies and Description Logic (DL) reasoning in mind, and hence comes with suitable OWL-specific plug-ins. HOBO comes with a generic 'Model Explorer' GUI that enables the model developer to browse the hybrid models and explore the dynamic behaviour of specific model instantiations. The HOBO framework provides the basis for an OWL-driven clinical data-entry application currently being developed in collaboration with an industrial partner.

Keywords: Ontology Driven Applications, Software Engineering, Software Frameworks

1 Introduction

Ontology-backed hybrid software models combine a domain-specific Object Model (OM), implemented in an object-oriented programming language such as Java, with one or more external models, expressed in suitable declarative ontology languages, such as OWL. The latter will typically come with associated reasoning mechanisms, such as Description Logic (DL) reasoners. HOBO is a successor to the generic hybrid modelling framework that was developed for the CLEF Chronicle system [1][2]. It is based on the same general principles, but is more flexible and provides a simpler and easier to use API for building the OMs. The HOBO framework is implemented in Java, as are the domain-specific OMs that it supports. Although totally independent of any particular ontology format, it was designed largely with OWL ontologies and DL reasoning in mind, and comes with suitable OWL-specific plug-ins.

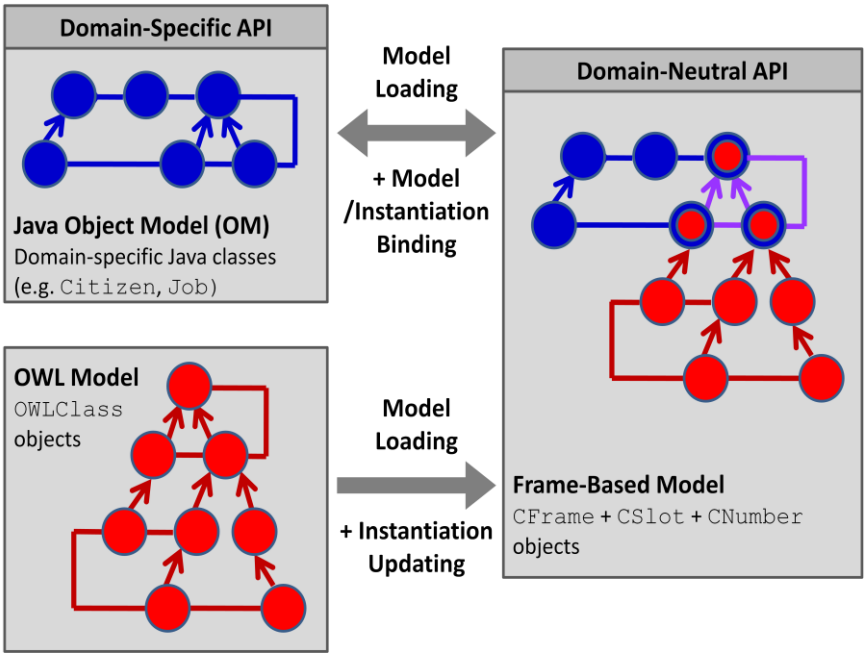
HOBO provides the core functionality for an OWL-driven clinical data-entry application currently being developed in collaboration with Siemens Healthcare, wherein it facilitates the integration of OWL with a dedicated modelling formalism developed specifically for the application. This architecture will be described in detail elsewhere. For the discussion in the remainder of the paper we will assume the simple case where HOBO is used together with OWL ontologies and DL reasoners.

adfa, p. 1, 2011.

HOBO, like its predecessor, provides both a domain-specific API (via the OM), and a domain-neutral API (via a dynamic Frames Model (FM) - described below). This enables client software to operate over whichever API is the most suitable for its particular purposes. Domain-specific client-code has been built on top of the Siemens clinical data-entry model, as implemented via HOBO, and a generic GUI-based 'Model Explorer' application has been built on top of the domain-neutral API (a tool that enables the modeller to (a) browse the hybrid model, and (b) create instantiations of specific concepts in order to explore the dynamic behaviour of those models).

A hybrid modelling framework such as HOBO that incorporates an OM has two main advantages over one based merely on OWL and DL reasoning. Namely (1) the domain-specific API associated with the OM makes the writing of domain-specific client code considerably simpler, and (2) the procedural processing capabilities of the OM enables the creation of models whose instantiations can dynamically update themselves in ways that cannot be achieved via DL reasoning alone.

2 HOBO Models



The above diagram provides an overview of the HOBO architecture. The central feature is a Frames Model (FM), which is dynamically generated at runtime, with the FM entities being derived from entities from either the OM or the ontologies, or, for some key sections of the model, both. The entities in the FM retain links with their original source entities, which are used in providing dynamic updating of the FM instantiations. As an instantiation is created by the client software it will be continu-

ously automatically updated in both shape and content by the HOBO framework. This dynamic updating is derived from (a) procedural processing by the OM, and (b) ontological reasoning via the DL reasoner. The links between the OM and the FM are tight two-way bindings. Hence the instantiation and subsequent updating of sections of the OM will automatically be reflected in equivalent operations being performed within the FM, and vice-versa.

2.1 Frames Model

The Java classes that the HOBO framework provides for representing the FM can be categorised by representational-level, as follows:

- **Concept-level:** `CFrame/CSlot/CNumber/CProperty`
- **Instance-level:** `IFrame/ISlot/INumber`
- **Meta-level:** `MFrame/MProperty`

FM instantiations consist of `IFrame/ISlot/INumber`-networks. These instantiation can be automatically updated via either DL reasoning or procedural processing by the OM. Updates can be either (a) addition or removal of frame-slots, (b) updating of slot constraints, or (c) addition or removal of slot-values.

The meta-level classes are used to specify value-types for slots whose values are concept-level entities. Values for both `MFrame`-valued and `MProperty`-valued types can be used in higher-order processing by the OM, whilst those for `MFrame`-valued slots can also be used in normal first-order DL reasoning .

2.2 Object Model

An OM is a domain-specific entity built from classes provided by the HOBO framework. These framework classes provide mechanisms for automatically generating the corresponding FM entities and setting up the required dynamic bindings between the OM and FM entities. Bindings are of the following types:

- OM classes/class-fields <==> `CFrame/CSlot` objects
- OM objects/object-fields <==> `IFrame/ISlot` objects

Hence, when an OM class is instantiated (i.e. when an OM object is created), the corresponding `CFrame` object along with its associated `CSlot` objects will be instantiated in the form of `IFrame/ISlot` objects. Similarly, whenever a `CFrame` is instantiated together with its associated `CSlot` set, then the corresponding OM class will also be instantiated. Once instantiated, any updates to one version of the instantiation, will be automatically reflected in the other.

2.3 OWL/DL-Based Plug-ins

HOBO comes with the following plug-ins specifically for loading OWL ontologies and reasoning over them with DL reasoners:

OWL Ontology Loading Plug-in: Loads OWL ontologies to create relevant sections of the FM. OWL constructs are used to derive concept-level frames and slots via a fairly obvious heuristic mechanism. The plug-in is configurable in various ways to specify which entities will be loaded, and how loaded entities will be interpreted.

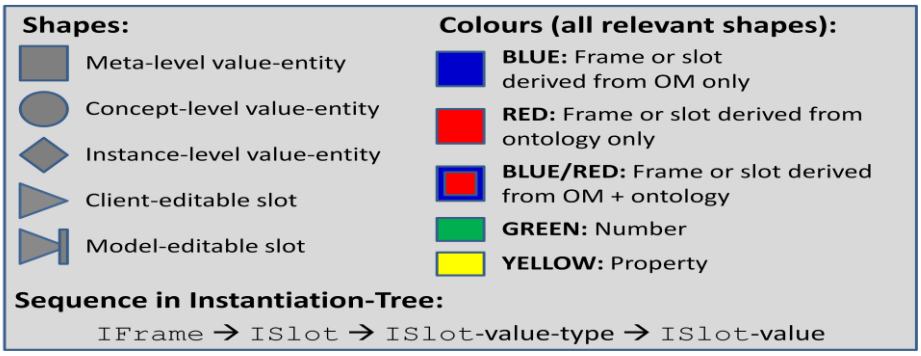
DL Reasoning Plug-in: Provides `IFrame` updates based on DL classification. The classifiable entities are derived from the current state of the `IFrame`, with configuration options including:

- *Entity type:* Either (a) instance-networks, or (b) class-expressions
- *Embodied semantics:* Either (a) open-world, or (b) closed-world (configurable on a per-property basis)

3 HOBO Model Instantiations

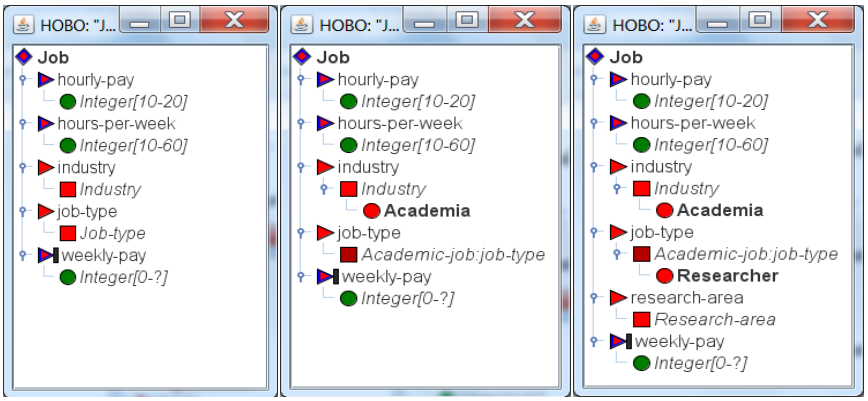
We now look at some examples that illustrate the dynamic behaviour of the model-instantiations. These examples are taken from a small model developed as a test model for HOBO, but designed to mirror certain features of the CLEF Chronicle model. Each example includes (1) a description of the behaviours illustrated, (2) a set of screenshots of the HOBO 'Model Explorer' GUI depicting the relevant sequence of user actions and subsequent model responses, (3) a table describing each action and response in this sequence (with columns aligned with the screenshots to which they refer). Though not illustrated here, all updates shown in all these example are completely reversible.

The following is a key to the icons used by the 'Model Explorer' GUI and shown in the screenshots.



3.1 Example 1: Simple updates via DL Reasoning

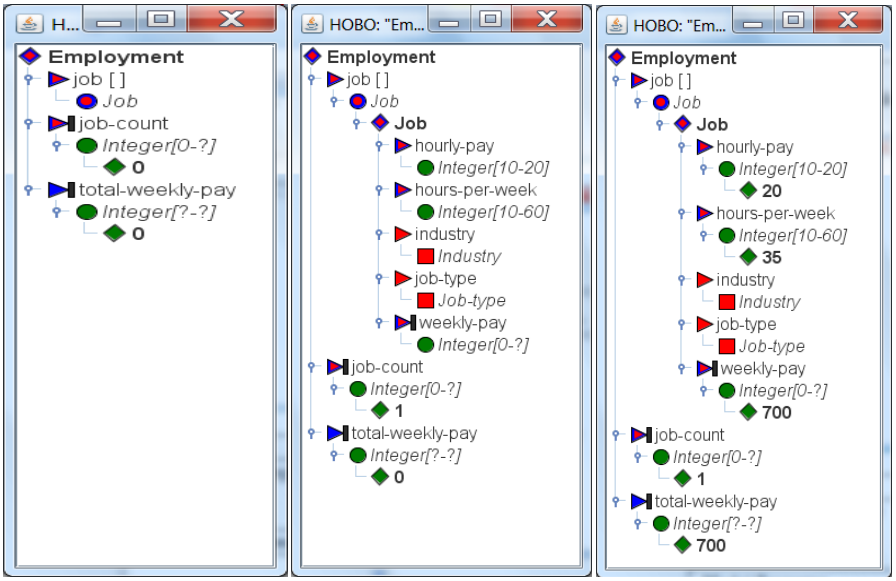
This example illustrates local updates to the model instantiation due to simple DL reasoning, showing how the setting of specific values for particular slots can cause (a) modification of constraints on other exiting slots, and (b) addition of extra slots to existing frames.



User Action	***User Action***	***User Action***
Instantiates `Job` concept	Sets industry slot to `Academia`	Sets `job-type` to `Researcher`
Model Response	***Model Response***	***Model Response***
	Constraint on `job-type` slot updated accordingly	`research-area` slot added

3.2 Example 2: Simple Updates by Object Model

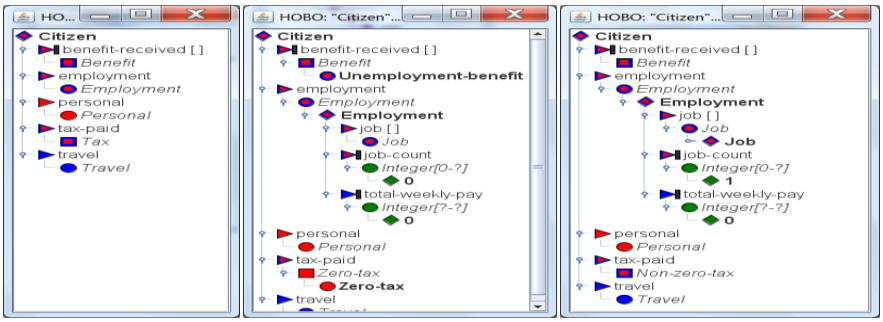
This example illustrates slot-value updates based on simple numeric calculations by the OM, including a trivial example of higher order processing (the value for the `job-count` slot is determined by counting the number of values in the `jobs` slot).



User Action	*User Action*	*User Action*
Instantiates `Employment` concept	Adds `Job` value to `job` slot	Sets values for `hourly-pay` and `hours-per-week` to
Model Response	***Model Response***	***Model Response***
`job-count` and `total-weekly-pay` slots set to `'0'`	`job-count` slot set to `'1'`	`weekly-pay` and `total-weekly-pay` slots set to appropriate values

3.3 Example 3: More Complex updates via DL Reasoning

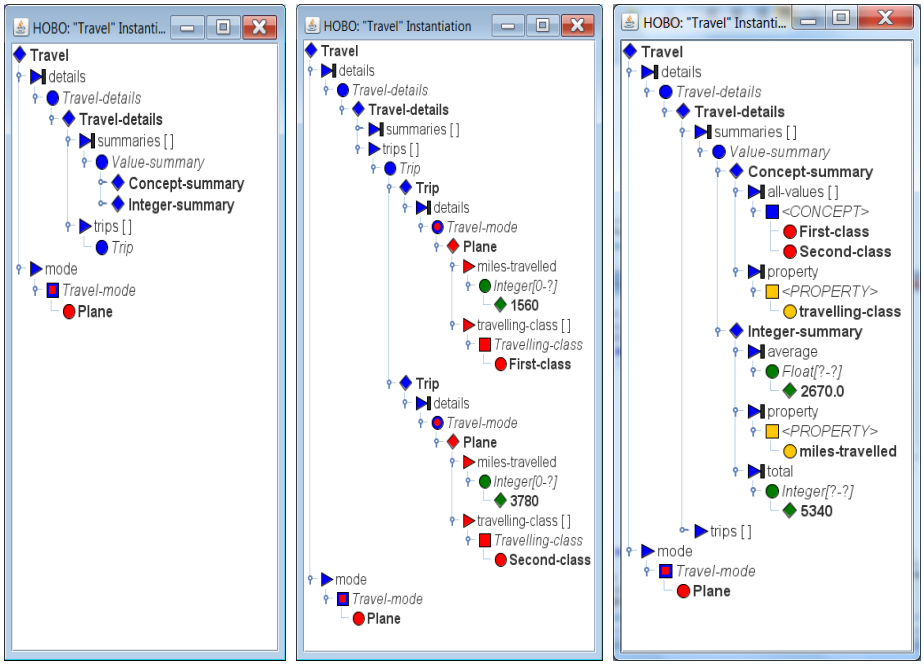
This example illustrates slightly more complex use of the DL reasoner, involving OWL expressions that are (a) nested, and (b) embody closed world semantics. The example also illustrates the automatic setting of inferred slot values. The close-world semantics are specified via the HOBO configuration file, which defines the `job` property as "closed", meaning that the generated OWL expressions involving this property will include suitable closure constructs. Thus when the `employment` slot is filled with an `Employment` frame for which no `job` values have been specified, values of `Unemployment-benefit` and `Zero-tax` are inferred for the relevant slots on `Citizen`.



User Action	*User Action*	*User Action*
Instantiates `Citizen` concept	Sets `employment` slot to `Employment`	Adds `Job` value to `job` slot
Model Response	***Model Response***	***Model Response***
	a) benefits-received slot set to `Unemployment-benefit` **b)** `tax-paid` slot set to `Zero-tax`	**a)** `benefits-received` slot cleared **b)** Constraint on `tax-paid` slot updated, and previous (now invalid) value removed

3.4 Example 4: More Complex Updates by Object Model

This example illustrates how the OM can dynamically create structure via higher order processing involving FM entities of which the OM has no built-in knowledge. Specifically, the example shows how a user-specified concept-level frame is used to create (a) a set of instance-level frames, and (b) a set of structures to summarise the values of the slots associated with the instance-level frames.



User Action	***User Action (centre screenshot)***
a) Instantiates `Travel` concept **b)** Set's `travel-mode` slot to `Plane`	Sets appropriate values for `miles-travelled` and `travelling-class` slots, for each `Trip` frame *[COLLAPSED IN RIGHT SCREENSHOT]*
Model Response	***Model Response (right screenshot)***
a) `travel-details` slot set to `Travel-details` **b)** `summaries` slot on `Travel-details` frame set to `Values-summary` **c)** `Values-summary` initialised for summarising the values of all slots on `Plane` *[COLLAPSED IN THIS SCREENSHOT]*	Appropriate summary values for each relevant slot added to structure under `summaries` slot *[COLLAPSED IN CENTRE SCREENSHOT]*

4 Related Work

Other than the CLEF Chronicle framework, the only system we are aware of that adopts a similar hybrid modelling approach to HOBO is Mooop [3]. The main ways in which Mooop differs from HOBO are (a) it is OWL-specific and does not abstract away from the underlying ontology format (b) it provides the client with greater access to the OWL semantics than HOBO does, and (c) it does not, as HOBO does, provide a domain-neutral API.

Most approaches to hybrid modelling seem to be based on code-generation. Examples are Sapphire [4], and the approach described in [6]. As far as we are aware, none of these types of approaches provide the kind of dynamic model-updating via DL-based reasoning at runtime that HOBO and Mooop do. Sapphire does however provide runtime mechanisms that "approximate the dynamic classification of OWL individuals".

5 Conclusion

From the discussion in this paper and the examples provided, it should be clear that HOBO provides rich facilities for building OWL ontology sensitive applications. In particular, it provides structured ways for "filling in the gaps" between the functionality OWL provides and the functionality applications need in a way that is natural to a Java programmer. As HOBO is pluggable, the same style of programming could accommodate domain models in other formalisms.

6 References

1. Puleston C, Cunningham J, Rector A. A Generic Software Framework for Building Hybrid Ontology-Backed Models for Driving Applications. *OWLED 2008.*
2. Puleston C, Cunningham J, Parsia B, Rector A. Integrating Object-Oriented and Ontological Representations: A Case Study with Java and OWL. *ISWC 2008*.
3. Frenzel C, Parsia B, Sattler U, and Bauer B. *Advanced Information Systems Engineering Workshops, CAiSE 2011*
4. Stevenson, GT, Dobson, SA. Sapphire: Generating Java Runtime Artefacts from OWL Ontologies , *ODISE* 2011
5. Kalyanpur A, Pastor D, Battle S, Padget J. (2004). Automatic Mapping of OWL Ontologies into Java. *SEKE, 2004*
6. HOBO software download page: http://owl.cs.manchester.ac.uk/research/topics/hybrid-modelling/