A Generic Software Framework for Building Hybrid Ontology-Backed Models for Driving Applications

Colin Puleston [1], James Cunningham[1], Alan Rector [1]

[1] Bio-Health Informatics Group, University of Manchester, United Kingdom.

Abstract. We describe a generic software framework for building dynamic domain-models that can drive both domain-specific and domain-neutral software applications. Models are hybrid. They combine standard ontological representations with extra-ontological representation to provide modes of interaction not fully specifiable within standard ontological formalisms. A core model-building framework has been developed along with some domain-neutral application software. A specifically temporal extension of this framework has been used in the representation of patient-record data. We use both this temporal extension and the patient-record use-case to illustrate the core model-building framework, the motivation for its development, and the rationale for the design decisions that it embodies.

1 Introduction

We describe a generic software framework for building dynamic domain-models that can drive both domain-specific and domain-neutral software applications. Models are hybrid. They combine standard ontological representations with extra-ontological representation to provide modes of interaction not fully specifiable within standard ontological formalisms.

Our model-building software comprises a fully generic 'Core Model-Builder' and a temporally-focused but domain-neutral 'Chronicle Model-Builder'. The latter is an extension of the former, specifically designed for building 'chronicle' models, concerned with the representation of richly-structured historical record data. It provides its own specialised extra-ontological representation combining:

- A 'SNAP/SPAN' representation [3], wherein the representation of time is split between point-like 'SNAP' events and temporally-protracted 'SPAN' events.
- A 'Temporal Abstraction System' that provides sets of 'temporal abstraction' [4] (i.e. historical summarisation) fields, and associated calculation procedures.

The motivation for the development of these model-builders was provided by the building of the 'Patient Chronicle Model' [1] for the CLEF project [2].

Model Architecture: The domain-models created via the model-builders (core or chronicle) embody an architecture consisting of two main sections:

- *External Knowledge Sources (EKS):* A set of ontological knowledge sources, which provide the detailed domain knowledge, together with any associated inference mechanisms required for the utilisation of that knowledge. Though potentially of varying formats, the EKS are currently in OWL [7].
- *Object Model (OM):* A dynamic Object Model, which provides both a static representation of some core domain concepts and a procedural specification of the required (non-ontologically-specifiable) interactions involving entities from both the OM and the EKS. The OM is implemented in Java.

The OM is the core component in this architecture, pulling together a set of one-or-more EKS together with their associated inference mechanisms. The OM represents only a relatively small number of fairly general core domain concepts, whereas the EKS provide the far more extensive detailed domain knowledge.

All access of the EKS by the OM is via a single transparent interface. The interface itself is very simple, much simpler than, for example, the OWL API [8]. However, it enables the OM to utilise the results of sophisticated 'under-the-hood' formalisms and inference mechanisms provided by the EKS (in our current setup this means OWL ontologies and Description Logic (DL) classification [5][6]). The OM requires no awareness of the actual contents of the EKS, other than an expectation that a handful of 'key-entities' be represented somehow and somewhere. The necessary mappings will be provided via a system of configuration files.

Model Building: The Core Model-Builder provides, via a Java API, the basic building-blocks for creating the domain OMs, interfacing with the EKS, and defining the required extra-ontological interaction. The Chronicle Model-Builder provides additional temporal-specific extensions, including a small set of pre-potted skeleton-patterns that form the basis for a specifically temporal type of extra-ontological interaction (which we illustrate in section 2).

Alternative Representations of Object Model: The OM provides a relatively simple means of specifying the extra-ontological interaction required by the model. On the other hand, this type of dynamic OM has a major draw-back in that it is not suitable for driving generic domain-neutral applications. In order to overcome this problem, the Core Model-Builder provides a facility for the translation of the domain-specific 'source' OM representation, into a derived 'network' representation, wherein the domain-specific class structure is replaced by a domain-neutral 'node'/'link'/'field' semantic-network-like representation. The translation process is automatic, with the specification being provided by the domain-specific classes themselves.

We can clearly see the advantages of this translation process by considering the example of a query-formulation application. If driven by the 'source' representation, it would need built-in knowledge of every class in the domain OM (classes such as *ProblemHistory* and *ClinicalRegime* for the Patient Chronicle Model). However, if driven by the 'network' representation, it only need deal with domain-neutral entities such as 'nodes', 'links' and 'fields'.

Model-Driven Software: We have developed the following generic software to operate over the 'network' versions of domain-models:

- An RDF-based data-storage system

- A query-formulation system.
- A SPARQL-based query-execution engine (with query expansion for both subsumption and transitive property relationships).
- GUIs for model-browsing, record-browsing and query-formulation.

Associated with the Chronicle Model-Builder are extensions to both the query-formulation system and the query-execution engine, concerned with dynamic temporal abstraction over arbitrary time-periods. We have also developed/are developing some domain-specific data-creation applications that operate on the 'source' representation of the Patient Chronicle Model.

In the rest of the paper we concentrate specifically on the Core Model-Builder. We focus on the chronicle-specific extensions (both the model-builder and the patient-chronicle use case) only in so far as they help illustrate the core framework. We provide no further details of the model driven software mentioned above. A technical supplement [9] looks at some of the wider aspects of the system.

2 Requirement for Hybrid Models

We will now look at the need for the type of hybrid model that the framework provides. Such a need arises when the dynamic behaviour to be modelled cannot be fully specified within the External Knowledge Sources (EKS). We illustrate this by looking at an example from the Patient Chronicle Model.

This example concerns the representation of the history of a clinical problem (such as cancer) displayed by a patient. Figure 1 provides a rough depiction of how this is represented in the Patient Chronicle Model. Two distinct types of object can be seen in this picture. Firstly, objects representing references to EKS-concepts for problem-type (e.g. 'Cancer') and problem-locus (e.g. 'Breast'). Secondly, instances of classes from the domain Object Model (OM), the main examples being:

- A *ProblemHistory* object representing the entire history of the problem.
- A set of *ProblemSnaphot* objects, each representing the problem at a specific point-in-time.
- A set of *DescriptorSnaphot* objects, one for each *ProblemSnaphot*, with each describing the problem at the relevant point-in-time, via a set of 'descriptors' derived from the EKS.
- A set of *DescriptorHistory* objects, one for each such descriptor, with each providing a summary of the descriptor-values over the relevant time-period. These summaries are expressed via a set of 'abstraction-attributes' derived from the Temporal Abstraction System.

The composition of the descriptor-sets and the current constraints on the individual descriptors will vary depending on the following factors:

- The current value of the problem-type field (cancer will not be described by the same set of descriptors as angina).
- The current value of the problem-locus field (breast-cancer will not be described by the same set of descriptors as lung-cancer).

The current values of the descriptors themselves (stage-II breast-cancer has a different set of potential sub-stages to stage-III and stage-IV breast-cancer, whereas stage-I breast-cancer does not have a sub-stage).

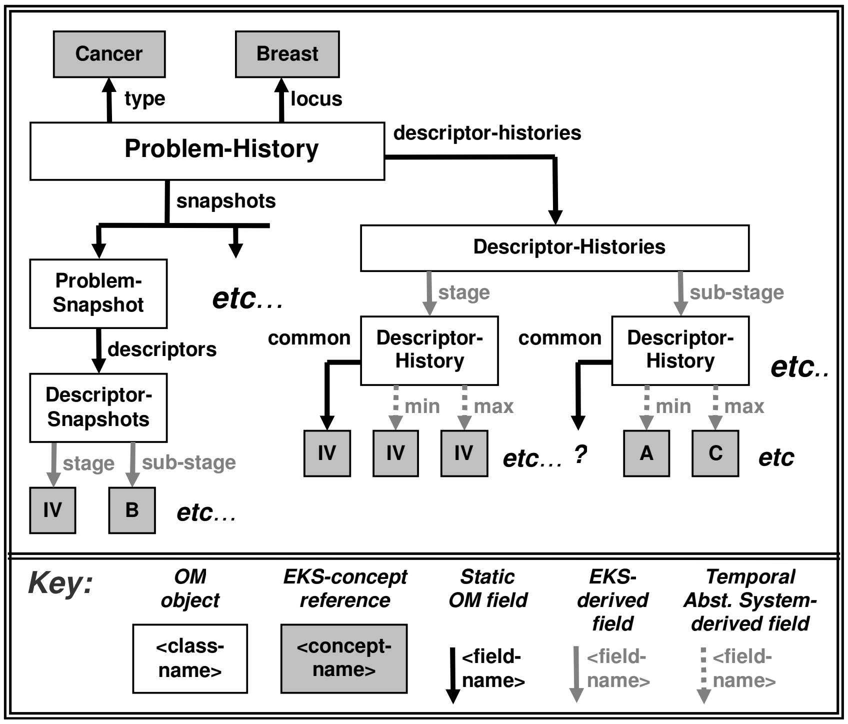

Figure 1: *Interaction Example from Patient Chronicle Model*

Constraints of this type can be expressed quite nicely in a language such as OWL. However, it can be seen that in our example there is no simple one-to-one correspondence between OM entities and the EKS entities to which they ultimately map. For instance, if the EKS is provided by an OWL ontology (as is currently the case for the Patient Chronicle Model), then as a value is specified for either the problem-type or problem-locus field, the following chain of events will occur:

- For each *ProblemSnaphot* object an OWL description is created, combining (1) the problem-type and problem-locus fields associated with the *ProblemHistory*, together with their current values, and (2) the set of descriptors associated with the relevant *ProblemSnaphot*, together with their current values. This description is classified using a DL classifier, and the results used to produce any relevant updates to the descriptor-set (including additions, deletions and constraint modifications).
- An analogous process is enacted involving the set of *DescriptorHistory* objects. In this case the problem-type and problem-locus fields are combined with the 'common-value' fields associated with the individual elements of this set, with the resulting classification leading to appropriate modifications to the

set. (The ‘common-value’ fields are special temporal-abstraction fields provided directly by the OM itself, and representing, if applicable, a ‘common’ value for the descriptor for the whole of the relevant time-period.)

- Any new *DescriptorHistory* objects created by this last action are provided with appropriate sets of ‘abstraction-attribute’ fields. These sets are derived from the Temporal Abstraction System, with their contents being dependent on the descriptor type (concept-valued, integer-valued, ordinal-valued, etc.).
- Certain abstraction-attributes (newly created or already existing) will have their constraints updated in accordance with the current state of the relevant ‘common-value’ field.

The important thing in all this is not the exact nature of the interaction, but the fact that such interaction requires a degree of orchestration not specifiable within the OWL ontology. More broadly, such higher-level interaction is not likely to be specifiable within any standard knowledge representation system, and certainly not one based on first-order logic. It is this consideration that provides the requirement for some form of hybrid model.

3 Object Model Architecture

We will now describe in more detail the architecture of the Object Models (OM) constructed with the model-builders, and then we take a brief look at how translation between the alternative ‘source’ and ‘network’ representations is specified.

Figure 2 shows the general architecture shared by all OMs (both ‘source’ and ‘network’ versions). This is a ‘conceptual’ architecture in that the OM components shown are not cohesive functional units, but are conceptual entities with their specification and implementation distributed throughout the domain OM. It should be noted that the means by which each of the OM components is specified/implemented varies between the ‘source’ and the ‘network’ representations, though we do not discuss the details here. In general, the OM classes perform two distinct functions: specification of the domain-model (via a combination of ‘Static’ and ‘Interaction’ Model), and the provision of additional procedural processing associated with the domain-model (via the ‘Additional Processing Mechanisms’).

Static Model: As specified via the API provided by the classes of the domain model. These classes implement certain generic interfaces, and conform to certain coding conventions. It is this that enables the version translation process to operate.

Interaction Model: Provides the specification of the interaction between the extra-ontological representation, any external sub-systems, and the EKS. As an instantiation of the model is built-up, the Interaction Model is continually kicking-in and updating the instantiation (as illustrated by the example in section 2).

Additional Processing Mechanisms: Procedural mechanisms associated with the model, which do not contribute to the shape of the model itself (and are therefore distinct from the processing provided by the Interaction Model). Examples include the derivation of additional data during model-instantiation (including the calculation of temporal abstraction values) and the derivation of on-the-fly temporal abstractions for the temporal extension of the query-engine.

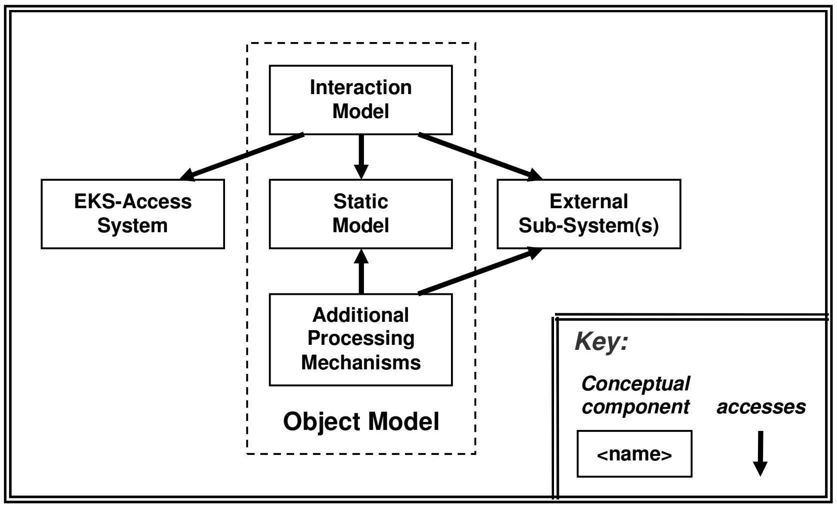

Figure 2: *Conceptual Architecture of Object Models*

3.1 Translation between Alternative Representations

Translation of an instantiation of the domain OM from ‘source’ to ‘network’ versions, involves (1) the translation of each object of domain-specific type into a ‘node’ with an associated set of ‘fields’ (2) the setting of appropriate field values to represent ‘links’ between nodes, references to EKS-derived entities and simple data-values. Translation can also be applied to the model itself. In this case the entire ‘source’ class-structure is translated into a collection of un-instantiated nodes (i.e. with no field-values set). The model is then constituted by the resulting collections of nodes and fields, plus the value-types and other constraints associated with the fields. It is this version of the model that forms the basis of the query-formulation system.

The main part of the translation is a fully automatic process by which the Static Model (or an instantiation thereof) is transformed into nodes and fields. This process is reliant on the domain classes implementing certain generic interfaces and conforming to certain coding conventions (e.g. each parameter-less public “get” method on a domain class will define a field on the relevant node).

Translation also involves the creation of a dynamic version of the Interaction Model in the form of a set of ‘listener’ objects attached to appropriate fields in the network. Both versions of the Interaction Model are defined by the individual domain classes, the ‘source’ version via appropriate procedural code which will kick in when required, and the ‘network’ version via the registration of an appropriate sets of ‘factory’ objects to create the required listeners at translation time.

There is also a reverse-translation process that enables the reconstitution of 'source' versions of individual model instantiations (obviously, reverse-translation is not applicable to the model itself). Reverse-translation of the Static Model is specified in a similar manner to forward-translation, via interface implementation and coding conventions. There is no requirement for reverse-translation of the Interaction Model, since it is provided by the classes of the reconstituted objects.

4 EKS Integration Architecture

The architecture responsible for the integration of the External Knowledge Sources (EKS) into the Object Model (OM) has been designed to achieve this integration in as seamless a fashion as possible. Specifically, we required that there be as few dependencies between components as possible, and that each component operate as generically as possible. Below we describe the individual components of this architecture (as shown in figure 3), but first we outline two central features upon which the architecture depends.

EKS-Incorporation Interface: Presents the contents of an EKS, and associated inference, in as simple a manner as possible, whilst providing access to arbitrarily complex representations and inference mechanisms. Implementations provide:

- A set of 'concept-hierarchies'
- A set of 'relationship-types' (used in handling constraints between those concept-hierarchies that are explicitly represented by the OM)
- Sets of 'descriptors' associated with individual concepts
- A 'concept-instance' creation function (which involves descriptor-set instantiation)
- A concept-instance state-update function (infers any required updates to the descriptor-set – the addition and removal of descriptors, and updates to constraints on existing descriptors)

The current OWL implementation of the interface involves the following mappings: concept $\leftarrow\rightarrow$ OWL-class, relationship-type $\leftarrow\rightarrow$ OWL-property, descriptor $\leftarrow\rightarrow$ OWL-property-restriction (interpreted via 'sanctioning' mechanism), concept-instance-state-update $\leftarrow\rightarrow$ DL-classification.

Key-Entities: The only point-of-contact between OM and EKS is via a handful of 'key-entities' that the OM handles explicitly, and which it expects to be represent somehow within the EKS. The OM will have no knowledge of how or where these entities are represented, or of the identifiers used within the EKS. There are only two types of key-entity, both of which correspond to entity-types represented by the EKS-Incorporation Interface: namely 'concept-hierarchies' (as defined via their root-concepts) and 'relationship-types'. To give an idea of the numbers involved: the current Patient Chronicle Model requires about fifteen of the former, and exactly two of the latter. The OM requires no knowledge of any of the vast majority of EKS entities that can be accessed via the interface. Hence, none of the individual concepts from the hierarchies, the descriptors associated with those concepts, or the individual constraints between the concept-hierarchies are represented as key-entities.

4.1 Architecture Components

The individual components of the EKS-integration architecture are shown in figure 3, which also shows the dependencies between the components, and categorises them by their level of awareness of the EKS. We will briefly describe each of the components in turn, and the roles that they play within the integration architecture.

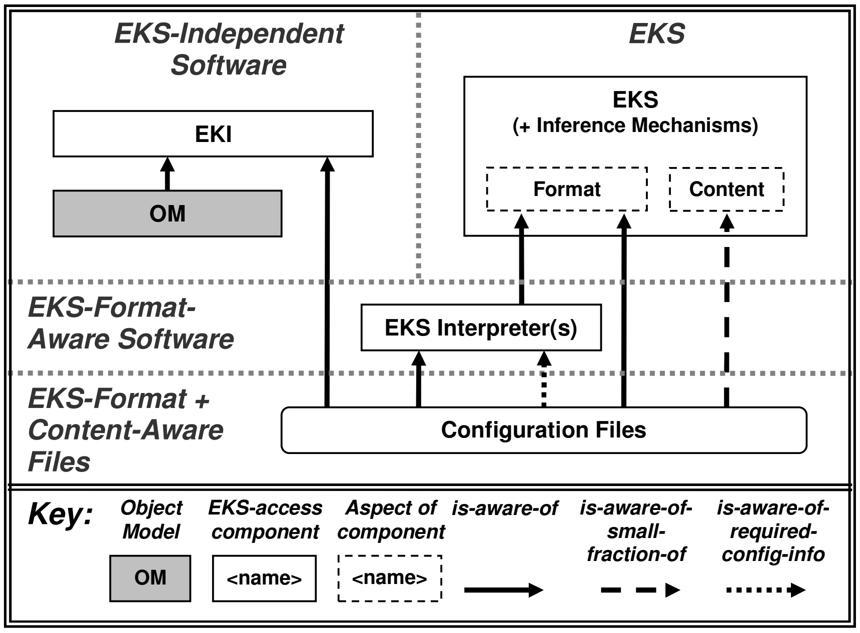

Figure 3: *EKS Integration Architecture*

External Knowledge Sources (EKS): The knowledge sources (OWL ontologies, etc.) being integrated into the system. As their name suggests, they are external to the system, and hence know nothing of any of the other components in the architecture.

External Knowledge Interface (EKI): A (Java) software package that provides a small number of very simple classes, each corresponding to one of the ‘key-entities’. These classes are used by the OM as its points-of-access to the EKS, with all relevant mapping information being provided via the configuration files.

Object Model (OM): Uses the classes provided by the EKI, firstly to reference the key-entities themselves, and secondly to access the detailed contents of the EKS, of which it has no prior knowledge (i.e. individual concepts from hierarchies, descriptors associated with those concepts, constraints relating to relationship-types).

EKS-Translators: Translation classes implementing the EKS-Incorporation Interface, one for each relevant EKS-format/conventions combination (there may be multiple different translators for a specific format, each recognising a different set of coding conventions).

EKS Configuration files: A system of configuration files, via which most of the connections within the architecture are specified. They provide the necessary information for the incorporation of each required EKS. This includes:

- Appropriate EKS-translator
- Mappings for all key-entities, between EKI class and EKS identifier
- Format-specific information to be processed by the relevant translator (e.g. location of OWL-file, configuration information for the DL classifier, etc.)

5 Discussion

We have described the motivation for the development of our model-building framework, namely the requirement for the building of domain-models that integrate both ontological and extra-ontological representations. We have also outlined the architecture of the models that this framework builds. We will now look at the rational behind the design of this specific architecture.

As we noted earlier, the type of interactions illustrated in section 2, and handled by the Interaction Model section of the Object Model (OM), is not of a type that can be represented in a standard declarative format, such as OWL. Hence, even if we were to represent the entire Static Model declaratively, we would still need supplementary mechanisms to handle the extra-ontological types of interaction. Therefore, if we take it for granted that we want to represent as much of our knowledge in as declarative format as possible, and that we should make as much use as possible of standard formats such as OWL, the need for some form of hybrid model follows.

The question of how best to provide such a hybrid model is not one with an immediately obvious answer. We are looking at a complex design issue with many potential solutions. However, we can state why we think the OM-based architecture provides at least a reasonably good solution, and outline the drawbacks of other broad categories of solution.

It is hopefully clear that to provide a declarative specification for the types of extra-ontological interaction with which we are dealing would be extremely difficult. The simpler option then, involved the integration of a procedural Interaction Model with a standard ontological style of representation. Our object-oriented approach has certain advantages here, namely that it:

- Allows for a single coherent representation of each concept, with all relevant knowledge, both static (the Static Model) and dynamic (Interaction Model), being incorporated into the definition of a single Java class.
- Provides a natural means of integrating any additional processing mechanisms (such as the calculation of ‘temporal abstraction’ values).

- Provides a domain-specific API for software such as the data-creation applications that need to operate with built-in domain knowledge.

Moreover, we avoid the main drawbacks of such an object-oriented approach by providing the alternative domain-neutral ‘network’ representation, and the associated translation system.

Whilst it would be difficult to classify all alternative approaches, it should be apparent that any attempt to replace the OM with something more declarative (such as OWL or UML) would be far from straightforward. An attempt to move the Static Model to such a format but to leave the Interaction Model as a procedural entity, would add complication, whilst making the representation of the core concepts less coherent, by dividing the definition of each into separate declarative and procedural sections. On the other hand, if we were to try to also represent the Interaction Model declaratively, we would require some form of supplementary formalism. This would not only add a great deal of complexity to the specification, it would also of necessity be less flexible than the procedural approach. Furthermore, it is not at all obvious that such an approach would actually buy us anything. In either case we would lose the additional benefits provided by the OM, namely the natural means of integrating additional processing mechanisms, and the domain-specific API.

In conclusion, our framework provides a relatively simple, elegant and flexible solution to a complex problem. Moreover, we find it difficult to envisage an obviously superior solution, although we would be interested in seeing any alternative approaches to similar problems. We would also be interested in finding other domains of application for both the Core Model-Builder (operating in combination with other types of extra-ontological representation) and the Chronicle Model-Builder.

References

1. Rogers J, Puleston C, Rector A. The CLEF Chronicle: Patient Histories Derived from Electronic Health Records. *IEEE Workshop on Electronic Chronicles (eChronicle'06)*
2. Taweel A, Rector, AL, Rogers J, Ingram D, Kalra D, Gaizauskas R, Hepple M, Milan J, Power R, Scott D, Singleton P. (2004) CLEF – Joining up Healthcare with Clinical and Post-Genomic Research. *Current Perspectives in Healthcare Computing*:203-211
3. Grenon P, Smith B (2004) SNAP and SPAN: Towards Dynamic Spatial Ontology. *Spatial Cognition and Computation* 4;1:69-104
4. Shahar, Y. and Combi. C. (1997). Temporal Reasoning and Temporal Data Maintenance: Issues and Challenges. *Computers in Biology and Medicine, 27(5), 353-368.*
5. Baader F, Calvanese D, McGuinness D, Nardi D, Patel-Schneider P. (2003) The Description Logic Handbook. Cambridge University Press. ISBN: 0521781760
6. Tsarkov D, Horrocks I. (2006) FaCT++ Description Logic Reasoner: System Description. *Proc of Int. Joint Conf. on Automated Reasoning (IJCAR~2006) (in press).*
7. W3C: Web Ontology Language (OWL) home-page. *http://www.w3.org/2004/OWL/*
8. OWL API home-page. *http://owlapi.sourceforge.net/*
9. A Generic Software Framework for Building Hybrid Ontology-Backed Models for Driving Applications (Technical Supplement). *http://intranet.cs.man.ac.uk/bhig/clef_misc/Ontology-Backed-Model-Builder-Supplement.pdf*