Integrating Object-Oriented and Ontological Representations: A Case Study in Java and OWL

Colin Puleston, Bijan Parsia, James Cunningham, and Alan Rector

School of Computer Science, University of Manchester, United Kingdom

Abstract. The Web Ontology Language (OWL) provides a modelling paradigm that is especially well suited for developing models of large, structurally complex domains such as those found in Health Care and the Life Sciences. OWL's declarative nature combined with powerful reasoning tools has effectively supported the development of very large and complex anatomy, disease, and clinical ontologies. OWL, however, is not a programming language, so using these models in applications necessitates both a *technical* means of integrating OWL models with programs and considerable *methodological* sophistication in knowing how to integrate them. In this paper, we present an analytical framework for evaluating various OWL-Java combination approaches. We have developed a software framework for what we call *hybrid modelling*, that is, building models in which part of the model exists and is developed directly in Java and part of the model exists and is developed directly in OWL. We analyse the advantages and disadvantages of hybrid modelling both in comparison to other approaches and by means of a case study of a large medical records system.

1 Introduction

A popular trend in software development is model driven engineering (MDE). In MDE, the primary artefact is not a program *per se*, but a model (which a program may instantiate). These models are typically expressed in a UML variant. Of course, programming languages, especially object oriented ones such as Java, themselves have modelling features and are often used to express (executable) models. The Web Ontology Language (OWL) [11] provides a modelling paradigm that is especially well suited for developing models of large, structurally complex domains such as those found in Health Care and the Life Sciences. OWL's declarative nature combined with powerful reasoning tools has effectively supported the development of very large and complex anatomy, disease, and clinical ontologies.

OWL, however, is not a programming language, so using OWL models in applications necessitates both a *technical* means of integrating OWL models with programs and considerable *methodological* sophistication in knowing how to integrate them. In this paper, we present an analytical framework for and evaluation of various OWL-Java combination approaches. We outline three distinct approaches to using ontologies to drive software architectures. The *direct* approach centres round the use of the ontological entities as *templates* for program classes. In this approach, the OWL based model is converted, statically, into a corresponding approximation in Java. The *indirect* approach represents the opposite extreme. In this case, the Java classes do not

model the domain concepts directly, but merely access an external model encoded in OWL. The third approach, presented in full in this paper, is a hybrid of the two. Here, the Java classes directly model a limited number of high level entities which are refined, dynamically, by aspects of the OWL ontology. The model is partly expressed by program classes and partly expressed by OWL classes but the two halves are integrated in a transparent way. The result is a model that can exploit the strengths of each side to compensate for weaknesses of the other, or to accommodate different skill sets and preferences of the modellers.

We developed the notion of a hybrid model, the supporting software, and the associated methodology in the course of the *Clinical E-Science Framework (CLEF)* project [11, 22, 33]. Part of the aims of CLEF were, broadly, to develop an architecture for representing series of *Electronic Patient Records* as coherent entities that capture a given patient's medical history in a unified form. We use the CLEF software [12] as the basis of a detailed case study of hybrid modelling.

Some key characteristics required of such a system drive us to hybrid modelling. First medical applications typically require a large knowledge base about medicine, e.g., disease, anatomy, treatment, *etc*. Given the specialised knowledge involved, the development and maintenance of this ontology needs to be performed by knowledge engineers with the requisite background and skills in medicine but who are not skilled software developers. The representation of this knowledge, and its use within the system, needs to be dynamic, in the sense that it can be modified or supplemented without any modifications being made to the software architecture that draws on it. Thus a good portion of the application's information is naturally modelled using logic based ontology languages (like OWL) common in the health informatics community.

However, capturing the types of complex temporally varying relationships that constitute a patient history, and, critically, doing this in a way that supports typical entry and searching patterns of the user base, is not a task for which OWL is particularly well suited. For this, complex data structures and procedures are needed within the model architecture. These requirements naturally suggested a combination of OWL, for its knowledge representation and reasoning functionalities, and an object oriented language, such as Java, for the procedural portions of the application.

In this paper, we examine and compare the three sorts of model especially for their distinct effects on software development. We show via a case study how a specific type of hybrid approach is well suited to a certain class of complex, information rich, dynamic applications.

2   Software Models

We now introduce our notion of what a software model is, and present a general framework to categorise the different varieties of software model.

Models: We use the term *model* specifically to mean a class based schema of some type that can be accessed via an API represented in some standard *Object-Oriented Programming Language (OOPL)*, such as Java or C++. The core of such a model is a hierarchically structured set of *classes* (not necessarily corresponding directly to specific classes in the host OOPL – see below), for each class an associated set of *fields*,
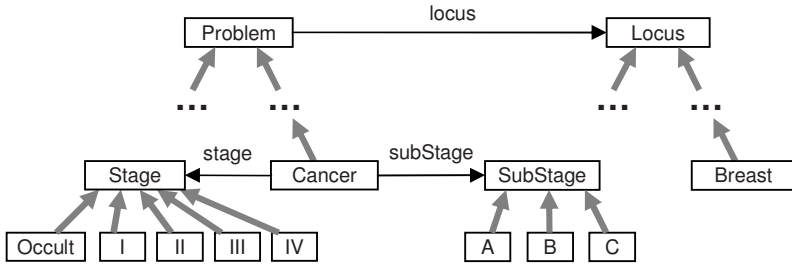
Fig. 1. Fragment of a simple software model - shows basic model entities (classes ▭ , fields ➤ , sub-class relationships ➤) whilst making no assumptions regarding mode of representation

and for each field a *type-constraint* defining the set of valid values. Figure 1 shows a fragment of such a model, concerned with the representation of patient problems, and specifically focusing on cancer and cancer-staging. Particular model formats may extend this structure in various ways, such as by providing cardinality constraints on fields, or providing data-type fields. Critically, the sorts of models we discuss are representations of a domain of interest, not (primarily) of the program itself. That is, the primary task is to represent the domain, not to structure the program.

Direct and Indirect Models (and Backing Models): One means of categorising such models is by the type of interface offered to the client code. There are two broad possibilities. A *direct* model is one in which the object model of the host programming language embodies the model directly, so that each OOPL class or field *is* a model entity. An *indirect* model is one in which the API presents the objects of a *backing*-model (BM) indirectly. Thus, instead of having a Java class called 'Cancer', an indirect model would use a generic Java class, say, ModelClass, a specific instance of which would then be used to represent a BM concept called 'Cancer'.

Figure 2 shows an instantiation of a fragment of an indirect model. A generic ModelInstance object represents an instance of a particular BM class, with the relevant class being specified via a ModelClass object (the current value of the instanceType field). Associated with the ModelInstance object are a set of model-fields that have been dynamically created, based on information derived from the relevant BM class. Each such dynamically-derived field will actually be represented via an object of an appropriate type (say, ModelInstanceField), with the current set of such objects providing the values for a single multi-valued OOPL field. We refer to such BM-derived, indirectly-represented model fields as *indirect* fields. We refer to model fields that are directly represented in the host OOPL as *direct* fields (such fields are found in both direct and hybrid models, but not in fully-indirect models).

An obvious difference between direct and indirect approaches is in their effect on the application development process. A direct model is tailor-made for a programmer writing domain-aware code, whereas an indirect model is more suitable for driving domain-neutral software. The converse usages are possible but more problematic. To drive generic software from a direct model requires the use of a reflection facility (as provided by the OOPL) in combination with appropriate coding conventions. Writing
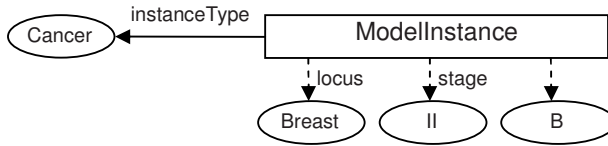
Fig. 2. Instantiation of a fragment of an indirect model – software entities represented both directly (object of named type ⬜, object field ⟶) and indirectly (reference to named model-class ⬭, model-derived field ⇢)

domain-aware software to operate over an indirect model is awkward and unnatural for the programmer, and hence inefficient (see section 3 for further discussion).

There are other issues that arise from the contrasting approaches, with pros and cons on either side. With indirect models the BM will generally be represented in a standard format for which sophisticated tools are available. For instance with OWL, several editors are available, such as Protégé 4 [13] and Swoop [14], and a range of reasoners [66] and other services. This is important when the model must incorporate a large amount of domain knowledge, and particularly, as is often the case, when the encoding is to be performed by a domain expert. On the other hand, direct representations provide a more natural means of implementing processing beyond the modelling formalism, to either contribute to the dynamic aspects of the model itself, or to operate over its individual instantiations.

An additional advantage of indirect models is in the possibility of BM encapsulation, which in addition to facilitating the seamless mixing and matching of disparate BM formats, also enables the filtering of BM constructs not relevant to the application. For example model classes may be generated only for certain types of concept (excluding for instance compositional concepts that play a role in reasoning but are not relevant to the application) and model fields generated only for certain properties (possibly identified via appropriate super-properties).

Dynamic Models: A *dynamic* model is one in which the details of the model can vary depending on the current state of the specific instantiation. The variability can be in the composition of the field-sets associated with specific instances, in the constraints on specific fields, or even in the types of specific instances. In general, the more dynamic the model, the more natural it is to use an indirect representation.

Figure 3 depicts the dynamic interaction involved in representing cancer staging, where the set of potential stages is dependent on the type of the cancer, and the set of potential sub-stages (if any), on a combination of type and stage. The modification of field constraints manifests itself in the re-setting of the default fillers. It is desirable that such automated updating be fully dynamic, with any assertions, retractions or replacements producing appropriate responses. For example, if the type of the disease is now specialised to Leukemia, the locus should update to Blood and the default stage value to Leukemia+stage, and the sub-stage field should disappear.

There are two basic ways of achieving dynamics in a model. Firstly, one can require the client code, after setting specific field values, to explicitly make any resulting updates to other parts of the instantiation, in line with a set of stipulations provided as part of the model. Alternatively one can create an update mechanism that
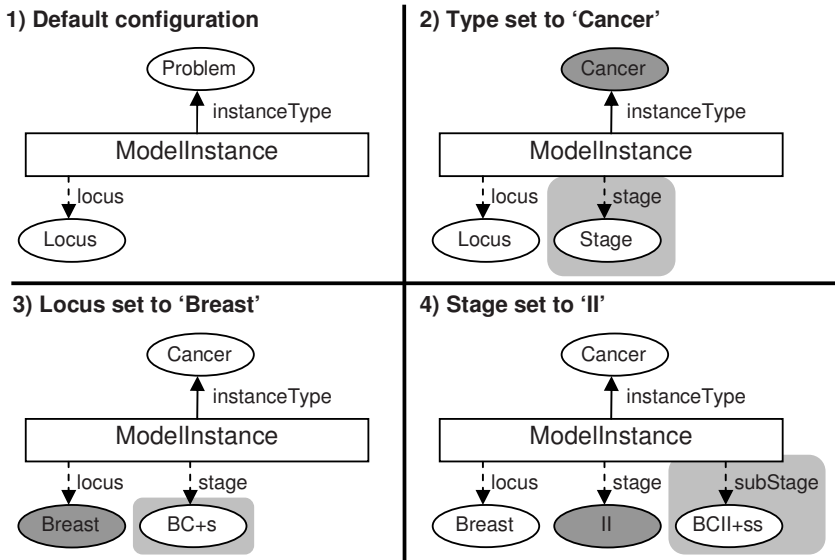
Fig. 3. Creation of an instantiation of a dynamic indirect model - basic key as for figure 2 - updates represented via shading (newly asserted field-value ⬭, area of automatic response by model ▢) - includes automatically-generated class names: BC+s = BreastCancer+stage = (I or II or II' or IV), *BCII+ss =BreastCancerStageII+subStage = (A or B)*

reacts appropriately to changes in model instantiations. The first alternative is the more flexible, imposing no restrictions on how the model is used. Hence, in addition to basic data-creation, the model could also be used to drive query-formulation, possibly with a variety of query schemas of varying expressivity. However, this flexibility comes at the price of additional complexity on the client side. Furthermore, the manner in which the updates are stipulated will be dependent on the BM format, ruling-out the possibility of BM encapsulation. The second alternative simplifies things on the client side, but does not necessarily provide the same flexibility. However, a suitable architecture can achieve the best of both worlds. For example, our framework provides an automatic update mechanism as part of an *instantiation building facility*. An associated *model-realisation plug-in facility* comes with alternative back-ends for data-creation and query-formulation. Additional back-ends (for e.g. alternative types of query-formulation) could be plugged-in if required.

Ontology-Backed Models: The types of dynamic model in which we are specifically interested are indirect models in which the BM is provided by an OWL ontology plus a suitable reasoner. In order for such a logic-based system to be used as the basis for a dynamic software model, some form of *sanctioning* scheme [88] must be used. Sanctioning provides a bridge between the constraint-based world of the ontology, and the field-based world of the software model. Specifically, a sanctioning scheme provides some means of associating a relevant set of fields with each OWL class. Exactly how this is achieved is not important here. Possible approaches include the use of heuristics to derive the field-sets directly from class
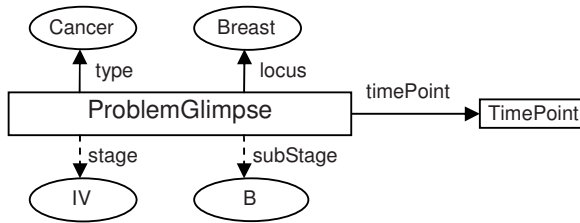
Fig. 4. Instantiation of a simple fragment of a hybrid model - software entities represented both directly (object of named type ▭, object field ⟶) and indirectly (reference to named ontology-concept ⬭, ontology-derived field ⇢)

restrictions, or the explicit specification of the field-sets via some form of internal or external meta-data.

Hybrid Models: We define hybrid models as software models that integrate both direct and indirect sections into a coherent whole. The intention is to benefit from the strengths of the respective approaches whilst mitigating their weaknesses. The hybrid models in which we are specifically interested are exemplified by the *Patient Chronicle Model (PCM)*, described in detail in section 4. Such models are divided into a direct core section, in which a relatively small number of core entities provide the main structure of the domain, and an indirect peripheral section, in which a far larger number of entities provide the detailed domain knowledge. The BM for the indirect section of the current version of the PCM is provided by an OWL ontology, though this is not a defining feature of such hybrid models.

To illustrate the basics of such models we look at an example from the PCM. Figure 4 shows the representation of a single disconnected "glimpse" of a patient's cancer at a specific point-in-time. It can be seen that this is a very similar set-up to that shown in Figure 2. Differences to note are: (1) the main entity is a domain-specific ProblemGlimpse object rather than a domain-neutral ModelInstance, (2) locus is a direct field on the ProblemGlimpse class (although stage and subStage are still dynamically-derived indirect fields), and (3) an additional timePoint field has been added (although the representation of time is a central feature of the PCM, and provides additional motivation for the use of a hybrid model, for the purposes of the current discussion we can consider timePoint as just another field). An additional difference (not depicted) is that the fillers for the concept-valued direct fields (type and locus) are actually of domain-specific types, designed to provide a type-safe means of representing references to concepts from the relevant section of the ontology. Hence, the type field has value-type ProblemType, a class that represents references to concepts from the Problem section of the ontology. (Note that in the PCM the mappings between the concept-referencing classes and the relevant root-concepts in the ontology are provided via a configuration file and are not hard-coded in any way.)

Figure 5 shows how collections of domain objects, such as ProblemGlimpse, can be aggregated together to form larger networks. The core structure of such a network is provided by the domain objects and their interconnections, or in other words, is the
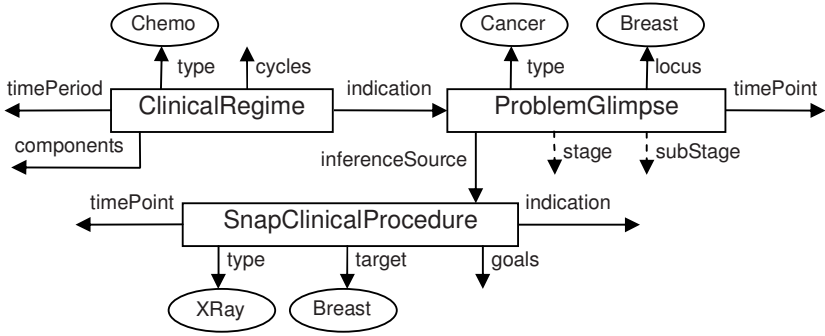
Fig. 5. Instantiation of a larger fragment of a hybrid model - key as for figure 4

instantiation of a direct model. It is only on the periphery of the model that the indirect elements intrude. On the other hand, the domain objects comprise only a small fraction of the model-entities – the vast majority residing within the ontology. For instance, the set-up in Figure 4 involves only three domain-specific classes, Problem-Glimpse, ProblemType and Locus (not to be confused with the Locus concept that provides the root of the hierarchy to which it maps), whilst the number of ontological concepts that can act as fillers for the type and locus fields, may number well into the thousands.

From the point-of-view of a programmer implementing a domain-aware application based on such a hybrid model, the direct nature of the core structure is a distinct advantage (as noted above in connection with fully direct models, and further discussed in section 3). However, the need for indirect model access has not been entirely eradicated. Providing fully direct access to the type of dynamic model with which we are dealing is simply not a practicable proposition. What the hybrid approach does do, however, is to greatly mitigate the problem by pushing the indirect representation to the edges of the model (in the case of the PCM, further mitigation is achieved by the provision of a dynamic model browser, which allows the programmer to explore the dynamic interaction in those areas of the model where it does need to be handled).

To provide a rough comparison of PCM-style hybrid models with both direct and ontology-backed indirect models, we have identified a number of potentially desirable features that the models may provide – see table 1. Although this set was derived directly from the requirements for the PCM, we feel that it is fairly comprehensive, though not necessarily exhaustive. Features include types of dynamic modification, as classified by modification-type (*field-constraint* or *model-shape - i.e.* the addition and removal of fields), and means of specification (*ontological* or *extra-ontological*). Also covered are type of API (*domain-neutral*, *domain-specific*), potential for attachment of processing mechanisms to operate over individual instantiations, knowledge maintenance by domain-experts, knowledge encapsulation, and potential for use in query-formulation. (*Note:* section 4 provides further discussion of some of the listed features.)

Table 1. Comparison of features offered by different types of software model

Feature	Direct	Ontology-Backed Indirect	Hybrid (PCM-style)
Dynamic modification (model-shape / ontological)	NO	YES	YES
Dynamic modification (model-shape / extra-ontological)	NO	NO	YES
Dynamic modification (field-constraints / ontological)	NO	YES	YES
Dynamic modification (field-constraints / extra-ontological)	YES	NO	YES
Domain-neutral API	YES[di1]	YES	YES
Domain-specific API	YES	NO	YES[hy1]
Model-instantiation processing	YES	NO	YES[hy1]
Knowledge maintenance by domain experts	NO	YES	YES[hy2]
Knowledge encapsulation	NO	YES	YES[hy2]
Query formulation	YES[di1]	YES[ob1]	YES
di1 = Given appropriate reflection based architecture ob1 = Given appropriate architecture hy1 = For core structure - not for detailed knowledge hy2 = For detailed knowledge - not for core structure			

Whilst both the fully-direct and fully-indirect approaches offer a subset of the listed features, PCM-style hybrids can, subject to certain trade-offs (see above discussion), be said to offer all of them. Obviously, when developing a software model one should select the approach that most closely meets the requirements of the particular domain, and where this implies a choice of options, one should probably go with the simplest. Hence, given the complexity overhead of the hybrid option, it should only be used if neither of the other options fits the bill. However, we feel that it is likely that for a large class of application areas this will indeed be the case.

3   Methodological Considerations

With a basic taxonomy of models in hand, we now turn to how various properties of the different sorts of model affect software development methodology via cognitive walkthroughs [10]. We consider how each sort of model handles the sequence of events shown in Figure 3, i.e. we (1) instantiate an instance of 'Cancer', (2) set the locus of the cancer to 'Breast', (3) set the stage of the breast cancer to 'II', and (4) set the sub-stage of the stage II breast cancer to some value. A key point about this sequence is that each setting of a field *alters* what other fields are available and the constraints upon those fields (and, thus, perhaps, the behaviour of the object in the application). Furthermore, the sequence of inputs and the particular values set (and thus the shape of the object) vary enormously. As described above, there are many different types of cancer, each with a different set of potential stages, and each combination of cancer and stage having a different set of sub-stages (or possibly none).

The "final" shape of the object (that is, when all its fields are set) is defined by a combination of (1) the model class of which the object is an instance, and (2) additional information provided by the model, specifying dynamic modifications specific to the evolving object. It is not always the case that there is a named model class that corresponds directly to the fully determined object (that is, the system can require runtime inference).

We take this sequence as an exemplar of how applications interact with entities in a model. It is easy to see that interactive applications (such as an electronic hospital chart) will need to modify entities in its model in this way. In all these cases, we presume that a sizable portion of the model will be expressed by a domain expert (who is probably not a programmer) in a suitable modelling language, such as OWL (*e.g.*, the specifics of cancer). Given this scenario, we can examine what costs and benefits model type offers. In order to keep things concrete, we confine the rest of the discussion to models expressed in OWL, Java, or a combination of the two, although the issues involved are potentially applicable to a range of modelling formalisms, and most general purpose OOPLs.

Consider direct models (*e.g.* as generated by an OWL2Java mapper [99]). One advantage of direct models is that the model is entirely captured in Java and the application programmer need never consider the ontology except as the input to the OWL2Java processor. Thus, the programmer can consider OWL to just be a funny kind of UML (and they may even view it as UML diagrams) and get on with the business of *using* the Java classes. If the ontology is small and unlikely to change, this is feasible. However, in typical medical applications, neither of these facts are the case. A recent version of the NCI Thesaurus contains 65,228 classes (up from 27,652 in 2003) and is updated monthly[1]. With a direct model, not only do we get a proliferation of Java classes that obscure the actual structure of the ontology, but the natural path to keeping the application in synch with the model is to regenerate and recompile[2]. Aside from the tedium of this procedure, it makes it practically impossible to modify the generated classes to introduce special behaviours, thereby eliminating a major benefit of the direct approach.

Furthermore, this sort of model is very difficult to work with given the sequence of operations in our example. In essence, to get the behaviour we want we need to determine in advance which specific class we are going to instantiate in step one (i.e., not just cancer, but *breast* cancer; and not just breast cancer, but *stage II* breast cancer; etc.). If we later want to change from stage II to stage I, or to correct the locus, we must discard our instance and create an instance of the relevant new sort.

Finally, since we do not have a reasoner available, we cannot query for aspects of the *ontology* that were not explicitly reflected into Java. Workarounds include trying to capture aspects of the semantics of OWL class expressions in Java (see [99]) or modifying the ontology to ensure that specific needed entailments get names, and are thus, reflected out to the application. In the first case, since the mapping is, at best, very partial and approximate, we still miss many possible entailments but now also get spurious ones. In the second case, we contaminate our model with various application

---

[1] http://nciterms.nci.nih.gov/NCIBrowser/Dictionary.do
[2] The Thesaurus is used in e.g.: http://cancerimages.nci.nih.gov/caIMAGE/index.jsp

specific classes and still cannot cover every case. Either way we would have great difficulty in replicating the type of behaviour illustrated by figure 3.

Of course all of this discussion concerning direct models assumes that the programmer is developing a domain-aware application. If alternatively the application is to operate in a domain-neutral fashion, the advantages described do not apply, whilst the difficulties in modelling dynamic behaviour are multiplied by the need to access that behaviour via some kind of reflection based mechanism.

In contrast, using an indirect model backed by an OWL ontology avoids many of these problems: The ontology is a separately modifiable component of the application. We have the full power of an OWL reasoner available and can even update the model in response to application events. Furthermore, the program does not have to incorporate thousands of classes, but only the small number of classes that provide the indirect model. As the API is domain independent, programmers can become expert in using that API and amortise the effort of learning it over many programs. Such APIs, as with SQL, provide a well defined interface for interacting with the ontology based model, so it is easy to analyse exactly where and how the application works with the model.

This flexibility can be accessed very nicely by a programmer developing a domain-neutral application, but for those developing domain-aware applications it comes at a considerable price. The indirect nature of the API becomes reflected in an unnatural indirect coding process, whereby the programmer must operate with API documentation in one hand, and some representation of the ontology in the other, to create code without type-safety or any other kind of API imposed constraints. Moreover, if parts of the behaviour required in the domain cannot be expressed in OWL (e.g., certain types of temporal relations, complex calculations, optimisations, *etc.*), the programmers must handle those aspects entirely on the Java side.

Of course, programmers could set up their own framework for mapping ontology classes into their Java based model on a case by case basis, but this is precisely building an *ad hoc* hybrid model where the details of the hybridization have to be managed explicitly.

With hybrid models we effectively split the difference. Consider the prototype scenario: the Java programmers developing the model can start with a high level model of the ontological aspects of the domain (say, a Problem class) as well as of other domain entities which do not appear in the ontology (e.g., a ProblemGlimpse). When the application is working with this abstract model the programmers can hook up a handful of key entities to corresponding entities in the ontology, in order to exploit the modelling done by the ontologist. Both the ontology and the program can dynamically modify the shape and constraints of their respective model elements without interfering with each other. Furthermore, programmers can naturally move the boundary between the part of the model which is in Java and the part which is OWL (or some other modelling formalism) as is appropriate. The fluidity of the boundary encourages programmers to use the formalism that is best for the job *given their tastes, experience, and skills*. Instead of having to jump directly into OWL (for example) they can defer that exploration until it is truly necessary. That is, *modelling* considerations, rather than limitations of their integration technique, drive shifts in the boundary. Obviously, hybrid models require *some* restrictions in the modelling on both sides.

All three approaches have sweet spots, though we believe that hybrid models are more generically useful. We suspect many domain applications will be better served by a hybrid model.

4   Case Study

We now look in detail at the *Patient Chronicle Model (PCM)*, as introduced in section 2, and at the generic framework with which it was built. The PCM provides the central component of an architecture designed for the representation of large bodies of patient record data in a richly-structured *chronicle* format, and their subsequent exploitation as a research resource over which interesting clinical queries can be formulated and executed. The framework comprises a fully generic *Core Model-Builder* and a temporally-focused but domain-neutral *Chronicle Model-Builder*[3], specifically for building chronicle-style models, such as the PCM. Although the framework was initially created with the PCM in mind, it is a generic entity that should have much wider applicability. Hence, our discussion here is concerned with the general class of hybrid model that the framework enables us to build, using the PCM for purposes of illustration. We concentrate on the external behaviour of the models. See [11] for a fuller description of their internal architecture. All of the software described here, including the PCM, the framework and the GUI-based tools, is available on the web (see [12]).

4.1   Design of Hybrid-Model Framework

An analysis of the requirements for the PCM resulted in the identification of the following set of elements to be incorporated into the design of the framework:

Ontological representation: An obvious requirement for a model representing medical data is that it in some way incorporates a large structured medical terminology. The fact that this terminology was required to support the type of representations and dynamic interactions illustrated in Figure 3 strongly implied some form of ontology with associated reasoning mechanisms. Furthermore, due to the size of the terminology, and the specialist knowledge that it was to embody, it was also necessary that the format facilitate maintenance by domain experts rather than software developers.

Temporal representation: Patient record data tends to come as a set of *snapshots*, representing such things as the current state of a patient's illness or the results of an x-ray procedure, at a particular point-in-time. However, to ask meaningful questions concerning a patient's *history*, we often need to aggregate together individual items of snapshot data into coherent entities representing, for instance, the entire history of a patient's condition. This implied a *SNAP/SPAN* representation [44] of some type, wherein the representation of temporal events is split between point-like *SNAP* events and temporally-protracted *SPAN* events. An associated requirement, to facilitate effective querying, was for the representation of temporal summarisations, or *temporal abstractions* [55] as they are known in the field of medical informatics. For instance,

---

[3] These components of the framework each comprise a set of classes and support utilities.

a set of measurements of the size of a tumour at various points-in-time can, with suitable interpolations, give rise to abstractions over selected time-periods, such as minimum-size, maximum-size, size-at-start-of-period, *etc*.

Temporal processing: Associated with the requirement for temporal abstraction structures, was a requirement for procedures to perform the relevant calculations. Also required was a *temporal-slicing* facility, for slicing SPAN objects up into sections representing arbitrary sub-periods. Such a facility is required for answering queries involving temporal-abstractions over dynamically-defined time-periods.

Ontological/temporal interaction: An additional requirement was for the orchestration of the higher-level interaction between the ontological representation, the SNAP/SPAN representation and the temporal abstraction structures (see below for details of such interaction).

Domain-specific API: The patient chronicle data is created programmatically by two data-creation applications, a heuristic-based *Chronicliser* that generates the richly-formatted patient chronicles from 'raw' patient record data, and a *Patient Chronicle Simulator* that generates realistic patient histories as an aid to system development. Both of these applications operate in a highly domain-aware manner and hence require a suitable domain-specific API.

Domain-neutral API: The model was also required to drive domain-neutral software, including an RDF-based *repository system*, with an associated *query-engine* (combining basic RDF querying with query expansion and dynamic temporal abstraction), and a set of GUIs for *model-browsing*, *record-browsing* and *query-formulation*. Hence, there was a strong requirement for a domain-neutral API (the alternative would be to let each application implement its own reflection-based interpretation of the model – obviously not a sensible option).

Query-formulation capability: The model was required to drive query formulation by domain-neutral applications, which, due to the dynamic nature of the model, implied a requirement for a flexible instantiation-builder with a model-realisation plug-in facility, to allow the incorporation of query-specific constructs into the instantiation (see discussion of dynamic models in section 2).

Fully-dynamic interaction: Since some of the model-driven software, such as the query-formulation system, needed to be highly interactive, it was required that instantiation-building be fully-dynamic in that the system respond appropriately to any assertions, retractions or replacements (see discussion of dynamic models in section 2).

Our hybrid model architecture, which was designed to incorporate this (partially conflicting) set of elements, is composed of (1) a central Java component incorporating both direct and indirect sections of the model, and (2) a set of one or more knowledge sources that collectively comprise the backing model (BM) for the indirect section. All BM access is via a clean API, which entirely encapsulates the underlying formalisms and associated reasoning mechanisms, allowing a range of formalisms to be mixed and matched. However, since the BM for the PCM currently consists of a single OWL ontology in combination with a FaCT++ reasoner [77] and suitable sanctioning mechanisms, we refer simply to the "ontology" throughout the following discussion.

4.2 Nature of Hybrid Models

We now look in more detail at the nature of the PCM-style models that can be built using our framework, building on the brief introduction provided in section 2. Specifically we look at (1) the type of complex dynamic interactions that the models can embody and the way in which they can exhibit behaviours not easily specifiable within a standard ontological representation, (2) the various distinct roles that procedural processing plays within the models, and (3) the 'network' representation, which enables both the models themselves and their individual instantiations to be accessed in a domain-neutral fashion.

Complex Dynamic Interaction: In section 2 we described how the ProblemGlimpse class can be used to represent a single disconnected 'glimpse' of a patient's cancer at a specific point-in-time (what we refer to as a *GLIMPSE* view). However, in practice ProblemGlimpse is used in only to represent transient conditions such as pain or headaches. For major conditions such as cancer, where we wish to track the progress of the condition through time, the more complex SNAP/SPAN-based representational pattern depicted in figure 6 is used. In this pattern, a series of ProblemSnapshot objects of the condition at specific points-in-time (the SNAPs) are aggregated together by a ProblemHistory object (the SPAN).

The first thing to note in this pattern is that the temporally-invariant type and locus fields are attached to the SPAN object, whereas copies of the temporally-variant stage and subStage fields are attached to each of the SNAP objects. This means that whereas the simpler GLIMPSE pattern could be mapped in a one-to-one fashion to a single ontological instance, the SNAP/SPAN version requires a collection of such instances, with each being mapped to a combination of the temporally-invariant fields on the SPAN object and the temporally-variant fields on a specific SNAP object.
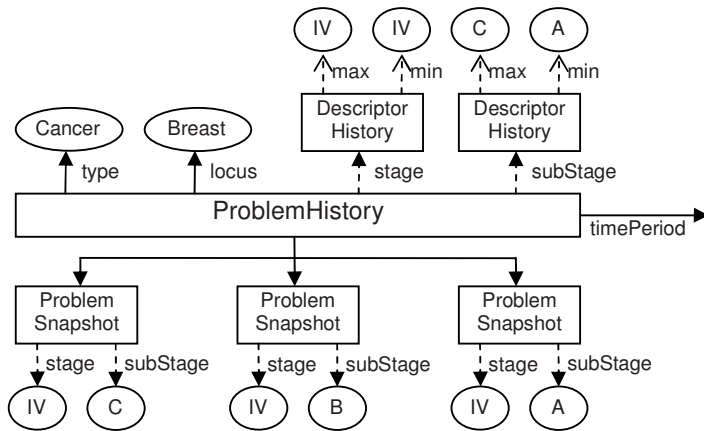

Fig. 6. Instantiation of a more complex fragment of a hybrid model (with problem description distributed between SPAN and SNAP entities) - basic key as for figures 4 and 5 - also show are fields derived from Temporal Abstraction System (- - ≫).(NOTE: the timePoint fields for the SNAP entities have been omitted.)

An additional element not present in the GLIMPSE version is the representation of temporal abstractions. For each current problem-descriptor (stage, subStage, etc.), there will be an abstraction field associated with the SPAN object, which provides an *abstraction-set* for that descriptor (via a DescriptorHistory object). Hence, the abstraction-sets for stage and subStage, both of which are defined (via extra-ontological meta-data) as ordinals, include attributes such as max and min. These abstraction-sets, as well as the methods for calculating the abstraction values, are ultimately provided by a *Temporal Abstraction System*, to which the PCM interfaces. This all adds additional complexity, involving (1) yet another ontological-instance, this one being mapped to a combination of the temporally-invariant fields on the SPAN object, and the set of abstraction fields, and (2) dynamic generation of the individual abstraction-sets, via interaction with the Temporal Abstraction System.

An important aspect of PCM-style hybrid models is the orchestration of interaction by the major domain classes within a representational pattern. For instance, locus has been specifically plucked out from the set of problem-descriptors to become a direct field on ProblemGlimpse (one reason being that it is an entity that the programmer will often wish to explicitly reference). However, even though the locus is explicitly represented in the direct model, it is still represented in the ontology (along with the more run-of-the-mill fields, such as stage and subStage, not represented in the direct model). Furthermore, we have seen that the locus value plays a part in the ontological reasoning behind the dynamic interaction illustrated in figure 3. Therefore classes such as ProblemGlimpse have to orchestrate the dynamic updating in a manner that maintains consistency between the model and the ontology.

With ProblemGlimpse, where there is a one-to-one correspondence between the entities in the direct model and those in the ontology, such orchestration adds nothing to the underlying ontological reasoning. It is merely the performance of a chore made necessary by the hybrid nature of the model. However, in the case of ProblemHistory where the corresponding interaction involves both multiple ontological-instances and the Temporal Abstraction System, its orchestration adds additional levels of complexity over and above that provided directly by the ontological reasoning.

Procedural Processing: Procedural processing plays three distinct roles within PCM-style hybrid models: *model-shape modification*, *field-constraint modification* and *model-instantiation processing*. Of these, the first two can be considered as providing an intrinsic part of the model, whereas the third acts on individual instantiations, but does not contribute to the model itself.

In the current PCM, the model-shape modification is always handled by fully generic mechanisms, although this is not something that is intrinsic to the task, and we could envisage a situation where shape modification of a more domain-specific nature was required. On the other hand, field-constraint modification and model-instantiation processing, as exemplified respectively by temporal abstraction and temporal slicing, are each, at different points in the PCM, handled by both generic and domain-specific mechanisms. The generic case can be seen from the ProblemHistory-centred pattern described above, where both the abstraction and slicing come as part of a configurable generic pattern. An example of the domain-specific case is provided by the DosagePattern class, used in representing sequences of drug administrations. This is an abstract base-class that provides both its own temporal abstraction fields (totalIntake,

averageDailyIntake, *etc.*), and its own temporal slicing facility. The actual processing is farmed out to appropriate sub-classes (RegularDosagePattern, CyclicDosagePattern, *etc.*), each of which provides a distinct way of either summarising, or directly representing the individual administrations. It should be apparent that the flexibility offered by the object-oriented core of the hybrid models is very useful here, whereas associating any sort of procedural processing with a fully indirect model is less straightforward, and the greater the required flexibility, the less appealing such an option becomes.

Fully-Indirect Representation: In order to provide the required domain-neutral API the framework embodies a mechanism for automatically translating the *source* version of the hybrid representation into a fully-indirect *network* version (and back again). The translation process depends on the Java reflection facility and the conformance of the source version of the model to certain coding conventions (the necessary ingredients for obtaining generic access to the direct sections of the model - as discussed above).

The basic translation operation takes a source domain class, such as ProblemHistory, and uses it to generate a set of generic network objects, consisting of a ModelNode plus a set of ModelFields. The generated objects will collectively represent an instance of the class. A wider process takes a model instantiation and converts it into an entire network. An additional mechanism is provided to enable the specification of the dynamic behaviour required from the network. This specification is handled by the individual domain classes, each of which, upon being loaded at run-time, can register a set of *factory* objects, which as the relevant sections of network are generated, are used to create sets of *listener* objects that will implement the required interaction.

The network representation can be used in two distinct ways. Firstly, to provide a static representation of an existing model-instantiation, which can be used in the storage, retrieval and browsing of records. Secondly, with appropriate extensions to represent the required query-specific constructs, as a dynamic query-formulation system. In this case the network representation is acting as an instantiation-builder with a model-realisation plug-in facility, in the manner described above. (The network representation could in principle also be used as an instantiation-builder for dynamic record-creation, but since this has not been a requirement of the PCM, our framework does not currently provide this facility).

5   Conclusions

OWL ontologies offer a number of modelling advantages that have been fruitfully exploited by domain experts working in Health Care and the Life Sciences and other areas. However, to reap the benefit of those advantages requires that the resulting artefacts (i.e., the models expressed as ontologies) are effectively exploited by programmers building applications. In this paper, we have presented three mechanisms for integrating OWL ontologies with programs written in a statically typed OOPL (specifically Java), including a new approach based on hybrid-models. Hybrid-models allow for a smooth integration between Java based modelling and OWL based modelling wherein each modelling paradigm's strengths can be mobilised as needed to produce a model that, on the one hand, is a reasonable representation of the subject

domain and, on the other, is a natural part of a program. We have shown that for a significant class of application this approach is especially effective.

Future work includes refining our hybrid-model supporting framework to better enable model refactoring and refinement. Right now, model development tools are entirely Java oriented or entirely OWL oriented, and thus do not allow for a unified view of the whole hybrid model. While one strong advantage of hybrid-models is that they allow different members of the development team to use the type of modelling technique that is most appropriate for the task or their own skill set, we believe that a holistic view of hybrid models has its own advantages.

References

1. Puleston, C., Cunningham, J., Rector, A.: A Generic Software Framework for Building Hybrid Ontology-Backed Models for Driving Applications. In: OWL Experiences and Directions Workshop (2008)
2. Rogers, J., Puleston, C., Rector, A.: The CLEF Chronicle: Patient Histories Derived from Electronic Health Records. In: IEEE Workshop on Electronic Chronicles, eChronicle 2006 (2006)
3. Taweel, A., Rector, A.L., Rogers, J., Ingram, D., Kalra, D., Gaizauskas, R., Hepple, M., Milan, J., Power, R., Scott, D., Singleton, P.: CLEF – Joining up Healthcare with Clinical and Post-Genomic Research. Current Perspectives in Healthcare Computing, 203–211 (2004)
4. Grenon, P., Smith, B.: SNAP and SPAN: Towards Dynamic Spatial Ontology. Spatial Cognition and Computation 4(1), 69–104 (2004)
5. Shahar, Y., Combi, C.: Temporal Reasoning and Temporal Data Maintenance: Issues and Challenges. Computers in Biology and Medicine 27(5), 353–368 (1997)
6. Baader, F., Calvanese, D., McGuinness, D., Nardi, D., Patel-Schneider, P.: The Description Logic Handbook. Cambridge University Press, Cambridge (2003)
7. Tsarkov, D., Horrocks, I.: FaCT++ Description Logic Reasoner: System Description. In: Furbach, U., Shankar, N. (eds.) IJCAR 2006. LNCS (LNAI), vol. 4130. Springer, Heidelberg (2006)
8. Bechhofer, A., Goble, C.: Using Description Logics to Drive Query Interfaces. In: DL 1997, International Workshop on Description Logics (1997)
9. Kalyanpur, A., Pastor, D., Battle, S., Padget, J.: Automatic Mapping of OWL Ontologies into Java. In: Software Engineering and Knowledge Engineering (SEKE), Banff, Canada, pp. 98–103 (2004)
10. Wharton, C.W., Reiman, J., Lewis, C., Polson, P.: The Cognitive Walkthrough Method: A Practitioner's Guide. In: Nielson, J.K., Mack, R.L. (eds.) Usability Inspection Methods. Wiley, New York (1994)
11. W3C: Web Ontology Language (OWL), http://www.w3.org/2004/OWL/
12. CLEF Chronicle Software: http://intranet.cs.man.ac.uk/bhig/clef_misc/chronicle/
13. Protégé 4 download page: http://www.co-ode.org/downloads/protege-x/
14. Swoop download page: http://code.google.com/p/swoop/