Ontology Driven Software Engineering for Real Life Applications

Michel Vanden Bossche[1], Peter Ross[2], Ian MacLarty[2], Bert Van Nuffelen[1], and Nikolay Pelov[1]

[1] {mvb,bvn,npe}@missioncriticalit.com
MISSION CRITICAL SA
Boulevard de France 9, Bat. A; B-1420 Braine-l'Alleud; Belgium
[2] {pro,iml}@missioncriticalit.com
MISSION CRITICAL Australia Pty Ltd
Level 23 HWT Tower; 40 City Rd; Southbank VIC 3006; Australia

Abstract. In this paper we introduce ODASE (Ontology Driven Architecture for Software Engineering). We present how we used ODASE to build a 250 person month e-insurance project for a multi-national insurance firm, where only 35% of the requirements were known at kick-off. We required one third of the time of the next closest quote for the project, and a similar project built classically at another insurance firm required also around three times the resources.

1 Introduction

In development of any large scale software systems the Business has a vision for a project to transform the company. However the vision is informally and incompletely specified and subject to frequent changes that leads to the Business-IT gap. This gap makes it difficult for IT to give a reasonable estimate of time and cost for the project and impacts greatly on the business case. Our experience shows that the gap can be bridged by describing the business knowledge in a formal language understandable by Business and consumable by computer programs.

The Web Ontology Language, OWL [5], was developed to facilitate greater machine interpretability of human knowledge by providing additional vocabulary along with formal semantics. The language forms a knowledge continuum between Business and IT, and provides a mechanism by which the Business can drive the evolution of the project by proposing concrete changes to the ontology.

By using the ontology as the "contract" between Business and IT, we delivered a large e-insurance project in 250 person-months. The project commenced with only 35% of the requirements fully specified. All the other bids were higher than 750 person-months, while some bid without any commitment to develop the key features which would discriminate the project on the market-place.

In the next section we introduce our Ontology Driven Architecture for Software Engineering, henceforth referred to as ODASE.

2 Development process using ODASE

The initial step in the ODASE development process is for representatives of the Business to work with a domain modelling expert and build a formal model of the

business. This model expresses all the business knowledge that is needed for IT to build the application. In our case the formal model is an OWL conceptual model. This step forces the Business to make requirements explicit, if the information needed can't be modelled or is extremely complicated to model it indicates instantly to the Business to think more about the feature they are trying to model.

At the end of the first step, one has an initial model of the business. This model can then immediately be evaluated by the Business adding example instances. This population of the model may also raise issues about the expressiveness of the model, and thus the model is iteratively refined.

We refer to the start of the ODASE process as creating a *knowledge continuum*. The ontology provides a "contract" about which both IT and the Business agree on the meaning of the concepts. It also provides a mechanism that allows the Business to formalize their specifications.

From this point on the actual application development starts. Here we focus on what we call the *construction continuum*: how IT uses the ontology to build the system. Our premise is that it is a waste of effort to formally express the specifications and *then* not use these specifications directly in the application. One of the main advantages of OWL is that it is a declarative language with a formal syntax and semantics. As such it can be used unambiguously by computer programs. The construction continuum is realized by transferring the knowledge expressed in the ontology to suitable objects, types and constructs in the programming language of choice.

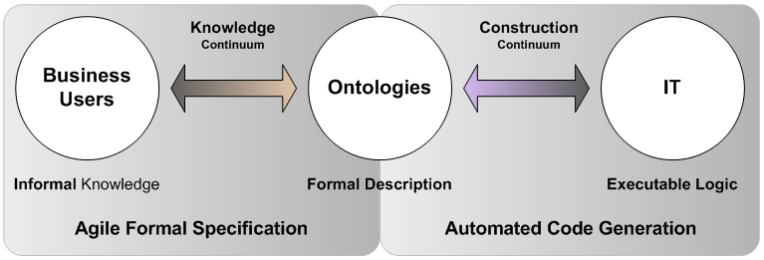

Fig. 1. Knowledge-Construction continuum

The process for transferring knowledge from the ontology to the programming language is by automatic generation of source code. For the ODASE platform, the underlying programming language is the Mercury language [6]. One of the advantages of Mercury is that it is a strongly typed programming language. This is critical for the ODASE development process because it makes it easier to propagate changes in the model to the rest of the application. When the model changes, the types will also change and the compiler will inform us of all the places in the code which are no longer consistent with the new model. Thus, the model can evolve without any concern of the code getting out of sync. This can also provide us with feedback on the amount of code changes required for evolving the model. This allows the Business to take informed decisions on

whether the evolution is on the critical path of the development, or should be done later, or maybe modelled an alternative way.

The choice of the Mercury language also has another advantage. As it is a logic programming language, OWL concepts can be represented in a natural way. For example, an OWL class C can be represented as a unary predicate C(x) and an OWL property P can be represented as a binary predicate P(x,y), where domain and range restrictions are taken into account by the Mercury types on the generated predicate type declaration, and cardinality restrictions are expressed as mode declarations. Also, when applying the necessary assumptions and language restrictions [1], the formal semantics of OWL and Mercury can be matched so that a *logical continuum* emerges from the business logic to the actual code (see Section 4).

3 Advantages of the ODASE approach

The key advantage of our approach is having a common agreement between Business and IT expressed by the ontology. As the Business is intimately involved in the construction of the ontology, they understand better the complexity of the development involved, thereby increasing the trust between both parties. When a requirement is difficult to express in the ontology, it gives a good indication of the complexity to the Business. Thus giving an early feedback to determine the trade-off between delay and functionality, a key requirement when time-to-market is critical.

Shifts in the technology landscape are frequent: mainframe, client/server, web 1.0, web 2.0, ... However, moving to a new technology is expensive because business knowledge is encoded in legacy systems. For example, a shift in the user interface technology might be difficult, because the business logic cannot be extracted from the technology dependent implementation.

Another key purpose in having a declarative description of the business knowledge is that it is possible to exploit the same information in different ways. For example, one can imagine a situation where validation rules are expressed in the ontology. The standard usage is to check whether a rule evaluates to true or false. Another usage is to determine for a rule that failed, the set of inputs causing the failure that are accessible on a user interface, allowing the corresponding fields to be highlighted on the screen. When the rule has been directly encoded in a program it is much more costly to provide this extra usage since a lot of source code must be adapted.

According to some studies [2, Fig. 1] around 80% of an IT budget is spent doing corrective and adaptive maintenance. In the e-insurance project, we reduced this risk by building interpreters for manipulating the information specified in the ontology. Since the code size of interpreters is in general smaller than code which has hard-coded the domain, less code is subject to potential bugs. The runtime performance cost of this approach was negligible compared to the increased flexibility which was required for being able to react quickly to new requests of the Business during the development.

During development, the Business populated the ontology with real data, while some parts of the application and the ontology were not completely specified. This allowed the Business to validate their requirements in the application at an early stage. A positive effect of this was that it allowed the Business to

identify weaknesses in the model early and consequently propose new concepts to capture the intended business meaning. Since the new concepts are formally expressed in the ontology, some adaptive maintenance is shifted from IT to the Business.

From an IT perspective, ODASE has the advantage that it relies on the formal semantics of the ontology, which means that major architectural changes can be made with clear and predictable effects.

4 The Hedwig platform

Hedwig[3] is the set of tools and libraries which we use to integrate an OWL ontology into an application. The Hedwig platform is central to the ODASE approach. It consists of several components that provide programmable access to information in RDF and OWL. The major components are: static type-safe access to ontologies, dynamic querying of ontologies, and a library for persistence of RDF graphs (memory, ODBC, Berkeley DB), all accessible via an abstract interface.

Hedwig supports various access patterns to ontologies. If both the ontology structure (T-Box) and the set of instances (A-Box) are static, Hedwig generates a type-safe representation of the ontology structure and all instances. If the ontology structure is static, but the instances can be updated at runtime, then a type-safe representation of the ontology structure is generated along with a declarative interface for updating the instances in the underlying RDF store. Finally if the ontology structure is dynamic — e.g. classes, properties and instances can be modified at runtime — the ontology structure and the set of instances are accessed through an OWL and RDF query language.

Due to lack of space we cannot provide a detailed comparison with other integrations in programming languages. Most existing approaches focus on the integration with Object Oriented programming languages (see [4] for general discussion). Most existing integrations provide either an abstract interface to the ontology or a type safe interface to an underlying updateable ontology store. In addition to these, Hedwig can provide a fully static (including instances) view on a static ontology. In that case we exploit the full potential of Mercury compiler to validate and optimize the code at compile time. This ability of our platform to offer both code generation and runtime query answering in a harmonised language is a crucial technology decision as our goal is to be able to serve a wide scope of applications with a maximum amount of automated code verification.

5 Future work

In our e-insurance application, a lot of the business rules and business computations were done in an internal domain specific language (DSL). The existence of the DSL meant that no integration with an off-the-shelve business rule engine was required. However in a completely green field implementation, a business rule engine would be required. We are currently investigating SWRL [3] as the

[3] Hedwig is a Mission Critical internal project and gets its name from the owl in the Harry Potter books by J.K. Rowling.

basis for the rule engine in our Hedwig platform. This will involve the development of a SWRL reasoning engine that is integrated with the rest of the ODASE platform.

6 Conclusion

The approach outlined by this paper was followed for an e-insurance project where only 35% of the requirements were known at kick-off. We have two ways of comparing our approach with current software development practices. The first is comparing the proposals for the e-insurance project, with the problem that all proposals are just estimates. The second is by comparing with a similar e-insurance application, with the problem that the functionality is not exactly the same. Comparing our approach with the other proposals we note that the project was completed with a third of the person months of the next closest quote. For a similar e-insurance application built classically for another insurance company, we also observed a factor of three difference for resource consumption.

This reduction in effort was due to the following advantages of the approach:

- common agreement between Business and IT;
- business knowledge never lost in a program;
- the same business knowledge representation is reusable for different purposes: e.g. consistency checking, automated error reporting, etc.;
- new model data added by the business with no IT interaction required which lowers costs;
- single language for domain experts and software engineers;
- single point of business definition;
- IT can rely on the formal semantics of the ontology;
- only new concepts require IT work;
- changes in the ontology are immediately reflected in the application;
- generating type-safe Mercury code from the business ontology made changes easier.

References

1. B. Grosof, I. Horrocks, R. Volz, and S. Decker. Description logic programs: Combining logic programs with description logic. In *Proceedings of 12th International Conference on the World Wide Web*, pages 48–57. ACM Press, 2003.
2. L. Hatton. Does OO sync with how we think? *IEEE Software*, 15(3):46–54, May/June 1998.
3. I. Horrocks, P. F. Patel-Schneider, H. Boley, S. Tabet, B. Grosof, and M. Dean. SWRL: A semantic web rule language combining OWL and RuleML. W3C Member Submission 21 May 2004.
4. H. Knublauch, D. Oberle, P. Tetlow, and E. Wallace, editors. A semantic web primer for object-oriented software developers. W3C Working Group Note 9 March 2006.
5. P. F. Patel-Schneider, P. Hayes, and I. Horrocks. OWL web ontology language semantics and abstract syntax. W3C Recomendation 10 February 2004.
6. Z. Somogyi, F. Henderson, and T. Conway. The execution algorithm of Mercury, an efficient purely declarative logic programming language. *Journal of Logic Programming*, 29(1–3):17–64, 1996.