Using SWRL for Rule-Driven Applications

Ian MacLarty[1], Ludovic Langevine[2], Michel Vanden Bossche[2], and Peter Ross[1]

[1] {iml,pro}@missioncriticalit.com
Mission Critical Australia Pty Ltd
22 Blackwood Street, North Melbourne, VIC 3051, Australia
[2] {llg,mvb}@missioncriticalit.com
Mission Critical SA
Boulevard de France, 9, Bât. A, 1420 Braine-l'Alleud, Belgium

Abstract. We relate our experiences using SWRL for commercial rule-based applications, and present a demo telco application where we used OWL to model the domain and SWRL to model the rules. We argue that using a formal, declarative rules language that operates over a formal and declarative model has distinct advantages over production rule systems that incorporate side-effects such as RIF-PRD, ILOG JRules or JESS. We also describe a simple, but effective implementation of our rules engine using the logic programming language Mercury and benchmark its performance. The rules engine is part of our ODASE platform for developing ontology-driven business-critical applications.

1 Introduction

When developing business software there is often a gap between what the business experts want and what IT thinks they want. Projects are often incompletely and informally specified and subject to frequent requirement changes. This leads to a "business-IT gap". This gap makes it difficult for IT to give reasonable time and cost estimates for the project. This problem is compounded as requirements inevitably change over time.

In our experience the business-IT gap can be bridged by describing the business knowledge in a formal language that is both understandable by the business experts and consumable by computer programs.

The Web Ontology Language, OWL [2], was developed to facilitate greater machine interpretability of human knowledge by providing additional vocabulary along with formal semantics. We use OWL to model the business domain and form a "knowledge continuum" between business and IT, providing a mechanism by which the business can drive the evolution of the project by proposing concrete changes to the ontology.

The Semantic Web Rule Language, SWRL [1], is a rule language designed to integrate closely with OWL. It supports adding Horn clauses whose atoms are OWL classes and properties. SWRL increases the expressivity of OWL and makes it possible to model more domain knowledge than OWL alone. We use SWRL to model parts of the business domain that are not easily or naturally modelled with OWL. The SWRL rules can be used in a production rule style where they are used to compute outputs, or they can be used for validation. SWRL builtins also provide a very natural extension mechanism whereby the modelling language can be enhanced with domain-specific builtins. For example in a bio-tech system we built, we modeled some chemical properties of genomics macro-molecules (such as DNA strain) in OWL. We then added special builtins for computing the melting temperature or the molecular weight of such bio-chemical compounds based on those properties.

In this paper we present our experiences using OWL and SWRL as the basis for rule-driven applications. We have written a simple proof-of-concept application, based on a real-world model in the telecommunications domain, that demonstrates some of the advantages of our approach. The demo is accessible online at `http://cloud.missioncriticalit.com/rule-demo`.

In addition to the above demo we have also developed a number of commercial systems using the same technology: a property and casualty e-Insurance system, an online financial planning and life insurance sales tool, an advanced sales process management tool and a web ordering system for the bio-tech industry. We have also developed a railway traffic management system which, although fully ontology-driven, is capable of supporting real-time events.

The rest of the paper consists of the following sections. In Sect. 2 we give an overview of our ODASE platform and our general approach to software engineering. In Sect. 3 we give an overview of the demo application. In Sect. 4 we discuss one of our SWRL engine implementations. In Sect. 5 we discuss the benefits of using SWRL as a rule language. In Sect. 7 we give some benchmarks showing that our SWRL engine is practical. Section 8 concludes.

2 The ODASE Platform

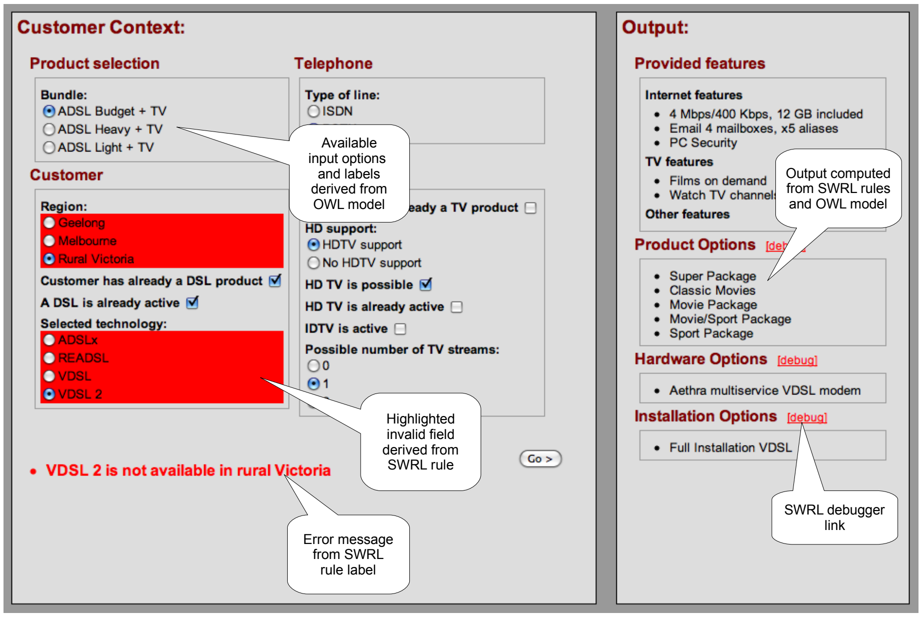
The demo application was build using our ODASE platform (Ontology Driven Architecture for Software Engineering) [7], [8].

In ODASE we use OWL and SWRL to model the business domain and Linear Temporal Logic to model business processes. Domain experts work with knowledge engineers to define the business model, rules and processes. Software engineers build generic interpreters that consume the model, rules and process specification to create a working application. Custom application-specific code (for example Java code in the UI) interfaces with the model through generated, domain-specific, type-safe APIs. This ensures that if the model changes, the custom code no longer compiles, forcing the code to be kept in sync with the model.

The generic parts of the ODASE platform are written in Mercury [5]. Using a logic language reduces the “impedance mismatch” between the modelling language and the programming language. Another reason for using Mercury is its strong type and mode systems which make it easier to engineer complex systems that are also robust and efficient. For example we are able to use Mercury’s type and mode system to guarantee that new domain-specific SWRL builtins do not have side-effects. Mercury can be compiled to C or Java. This makes it easy to integrate ODASE with Java applications.

3 The Demo Application

The application shows the options available to a user who wishes to purchase a broadband internet/TV package, based on relevant information about the user. An annotated screenshot of the demo is shown in Fig. 1.



Customer Context:

Product selection

Bundle:
● ADSL Budget + TV
○ ADSL Heavy + TV
○ ADSL Light + TV

Customer

Region:
○ Geelong
○ Melbourne
● Rural Victoria

Customer has already a DSL product ☑

A DSL is already active ☑

Selected technology:
○ ADSLx
○ READSL
○ VDSL
● VDSL 2

Telephone

Type of line:
○ ISDN

eady a TV product □

HD support:
● HDTV support
○ No HDTV support

HD TV is possible ☑

HD TV is already active □

IDTV is active □

Possible number of TV streams:
○ 0
● 1

- VDSL 2 is not available in rural Victoria

Output:

Provided features

Internet features
- 4 Mbps/400 Kbps, 12 GB included
- Email 4 mailboxes, x5 aliases
- PC Security

TV features
- Films on demand
- Watch TV channels

Other features

Product Options

- Super Package
- Classic Movies
- Movie Package
- Movie/Sport Package
- Sport Package

Hardware Options

- Aethra multiservice VDSL modem

Installation Options

- Full Installation VDSL

Available input options and labels derived from OWL model

Highlighted invalid field derived from SWRL rule

Error message from SWRL rule label

Output computed from SWRL rules and OWL model

SWRL debugger link

Fig. 1. Annotated demo screenshot

The screen is divided into two halves. The first half of the screen contains questions about the user which are used to determine what options are available, while the second half displays the currently available options. The input section is modelled as a *CustomerContext* class in the ontology. The *CustomerContext* captures technical details of the customer's installation that may have an impact on the offered product configurations (e.g. localization and technological support available in the area). The *CustomerContext* includes information like what sort of package (bundle) the user is interested in, where the user lives (region), what technology the user already has installed and what technology is possible for that user. In the full application some of the *CustomerContext* would be populated with values from the telco's backend systems, rather than via the UI.

The available options are derived from rules in the model.

The OWL model contains 79 classes, 86 properties and 156 instance. There are 21 SWRL rules. It is not a very large ontology, but there are some complicated relationships in the model (Pellet 2.0.0 takes 11 minutes to classify it on a 2 GHz Core 2 Duo machine).

The model captures the business domain of a telco company. This includes marketing aspects such as which services and options are offered to the customer and how they are packaged into products. The model also includes technical constraints that link the services, the installation and the hardware.

For example the *Offer* class models all the products a customer can purchase through the application. Among them, the *Bundle* offers aggregate several offers into a single one. The *Feature* class represents all the elementary services that may be seen by the customer. This class admits several subclasses such as *InternetCustomerFacingService* (the various internet accesses and options) or *Installation* (the set of all the installation methods which can be proposed to activate an offer). The *provides* property links an offer to the features it includes. The following rule states that all the features provided by an offer are also provided by the bundles that aggregate this offer:

$$\mathit{Bundle}(?b) \wedge \mathit{aggregates}(?b, ?o) \wedge \mathit{provides}(?o, ?f) \rightarrow \mathit{provides}(?b, ?f)$$

The demo runs as a Java servlet. We compile the Mercury code to Java so that it can run in the same Java Virtual Machine (JVM) as the servlet.

The user interface is written with Apache Wicket, a Model-View-Controller orientated Java web framework. We have a tool that generates a bean-like domain-specific Java API for the OWL model. For example it generates a Java class called `Offer` with a method called `getProvides` that returns a list of objects of the class `Feature`. The generated `getProvides` method calls the ODASE SWRL/OWL engine which is implemented in Mercury. This makes it possible to integrate the OWL model with the Wicket frontend (or any other Java application) in a type-safe way.

4 The Rules Engine

In ODASE we have several rules engines designed for different tasks. The engines all implement a common interface so that they can be easily interchanged and stacked on top of each other. The engines can access various kinds of data sources, including SQL databases or a in-memory stores.

For the demo application we use an in-memory store. The SWRL engine uses backward-chaining with minimal model tabling. The algorithm is a modified version of OLDT resolution [6]. Note that the engine interprets the SWRL rules directly and does not require translation of the SWRL rules to Mercury code.

An important benefit of using backward-chaining is that we only query the underlying facts store (which could be a large SQL database) for facts that are relevant to answering the query at hand and we don't need to know what facts will be required beforehand and put them into working memory. Contrast this with a rules engine such as JESS which uses the RETE algorithm and requires working memory be populated with all relevant facts before executing the rules.

We reason about OWL axioms by translating the OWL axioms into SWRL rules and interpreting them with the SWRL engine. For example we translate the OWL axiom *TransitiveProperty p* into the SWRL rule:

$$p(?a, ?b) \wedge p(?b, ?c) \rightarrow p(?a, ?c)$$

Note that if we used normal SLD resolution instead of OLDT resolution such rules would cause infinite loops.

It is well known that not all OWL axioms can be fully translated into SWRL rules. This means that the resulting SWRL program can be incomplete with respect to the OWL semantics of the model. Generally we have not found this to be a problem in practice. In our experience the business model can usually be translated to SWRL without loss of completeness.

5 Benefits of Using SWRL

SWRL is a declarative rules language not bound to a particular execution algorithm. Instead a formal model-theoretic semantics is defined which determines when an execution algorithm is sound (gives correct results) and complete (gives all results). Executing a SWRL rule does not have any side-effects, such as updating a database, sending an email or selling shares.

Most currently available rule languages (such as ILOG JRules, JESS or RIF-PRD) are tied to a specific execution algorithm (usually RETE-based). Rules written in these engines often have side effects. This is particularly true when the rules can access Java objects directly (as is the case with JRules and JESS). The called Java methods can execute arbitrary code which could have side effects. This means that these engines cannot use different evaluation strategies, because the order in which the rules are executed matters. This complicates the task of writing rules, because it means the rule author needs to be aware of any side-effects the rules might have as well as the algorithm that is used to execute the rules. The rule author may also need to assign priorities to rules to control the order in which rules are fired. This all makes rule authoring more of a technical task than a business (domain expert) task, which negates one of the purported benefits of using a rules engine: that business knowledge is separated from technical IT knowledge and that the business itself has more direct control over the behaviour of the application. In our experience this separation of business knowledge and IT knowledge can only be achieved if the business knowledge is encoded declaratively. This decouples it from the technical implementation of the rules engine.

Having a declarative rule language that is independent of the evaluation algorithm also has other benefits. One benefit is that IT can choose the most efficient evaluation algorithm for the particular domain. IT can also change the algorithm at any point without having to change the business rules.

Another benefit is that the business knowledge encoded in the rules can be re-used in different and interesting ways. For example in one application we wrote, we would use the rules to work out what questions to ask the user to satisfy a certain goal. To do this we interpreted the rules using an algorithm that worked out what properties needed values to satisfy a particular goal. We then automatically generated fields in the user interface for those properties.

SWRL rules can also be used for validation. For example in the demo application we have added the following rule:

$$\mathit{technology\text{-}type}(?\mathit{context}, \mathit{VDSL2}) \land \mathit{region}(?\mathit{context}, \mathit{rural\text{-}victoria}) \rightarrow$$

This rule has an empty head. An empty head corresponds to false. The rule is thus invalid if the body is true (a rule is invalid if the body is true and the head is false). In plain English the rule says that VDSL2 is not available in the rural Victoria region.

In the demo this rule can be triggered by selecting VDSL2 and Rural Victoria and clicking the “Go” button. You will notice that an error message is displayed and the “Selected Technology” and “Region” fields are highlighted to indicate that inconsistent values were selected for those fields (see Fig. 1). The error message comes from the rule’s label in the ontology. The application automatically works out which fields to highlight by analysing the invalid rule. It effectively builds a proof tree for the body of the invalid rule and compares that to the fields that are currently being displayed to the user. If any of the currently displayed fields correspond to atoms in the proof tree, they are highlighted.

The same validation rule could also be expressed using a production rule system that integrates directly with a Java model (such as JRules). It would look something like the following:

```
if
    context.getTechnologyType() == VDSL2
    and context.getRegion() == RURAL_VICTORIA
then
    showErrorMessage("VDSL2 is not available in rural victoria")
    highLightField("technology-type")
    highLightField("region")
```

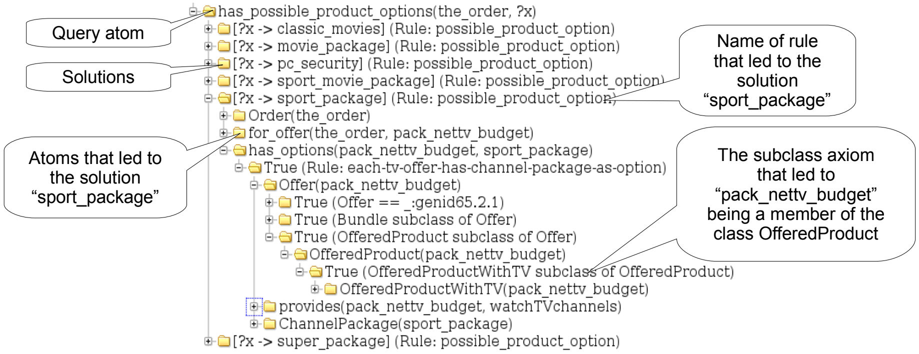
The problem here is that we have to encode user interface information in the business rule. We cannot evaluate the rule a different way to work out what properties in the model caused the error message to be displayed. This conflates business knowledge and user interface programming and makes the job of maintaining the rules more onerous.

The last benefit of using SWRL that we will mention here is its tight integration with OWL. SWRL allows one to add rules that work directly with the concepts and relationships defined in the OWL model. There is no impedance mismatch between the modelling language and the rules language. JRules, JESS and RIF-PRD require the business model to be mapped to an intermediate language that the rule engine can interpret. This adds extra work for the maintainers of the rules system, introduces another source of potential errors and makes it harder for domain experts to add arbitrary new rules.

6 Debugging

Since the declarative semantics of SWRL are very similar to those of pure Prolog, we can use well known declarative debugging techniques to debug the rules [3]. Declarative debugging is particularly suited to debugging business rules systems, because it allows the rules to be debugged without having to know the intricacies of the execution algorithm. This makes it easier for domain experts to debug the rules. We have included some simple online declarative debugging facilities in the demo. Clicking the “debug” link that appears next to some of the outputs will start up the debugger. A screenshot of the debugger is shown in Fig. 2.

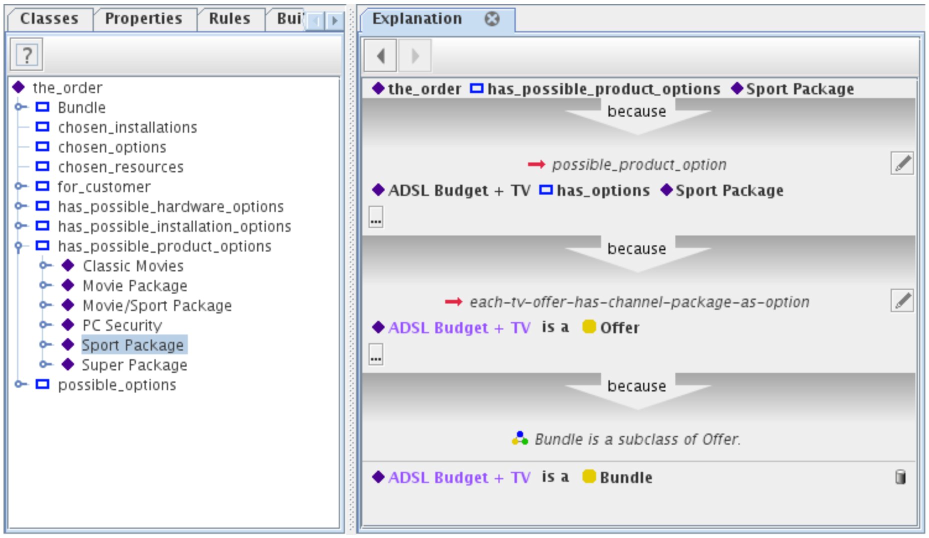
The debugger shows a lazily constructed proof tree of the computed answer (since we use OLDT resolution the proof tree could be infinite, so it must be lazily constructed). There are two types of nodes in the proof tree. The first type represents a query. The children of a query node are the computed solutions to the query. The rule name or OWL axiom that generated the solution is given in parenthesis after the solution. The children of a solution node are the queries used to directly compute that solution. Typically these correspond to the atoms in the body of the rule that generated the solution.



```
⊟ has_possible_product_options(the_order, ?x)
  ⊞ [?x -> classic_movies] (Rule: possible_product_option)
  ⊞ [?x -> movie_package] (Rule: possible_product_option)
  ⊞ [?x -> pc_security] (Rule: possible_product_option)
  ⊞ [?x -> sport_movie_package] (Rule: possible_product_option)
  ⊟ [?x -> sport_package] (Rule: possible_product_option)
    ⊞ Order(the_order)
    ⊞ for_offer(the_order, pack_nettv_budget)
    ⊟ has_options(pack_nettv_budget, sport_package)
      ⊟ True (Rule: each-tv-offer-has-channel-package-as-option)
        ⊟ Offer(pack_nettv_budget)
          ⊞ True (Offer == _:genid65.2.1)
          ⊞ True (Bundle subclass of Offer)
          ⊟ True (OfferedProduct subclass of Offer)
            ⊟ OfferedProduct(pack_nettv_budget)
              ⊟ True (OfferedProductWithTV subclass of OfferedProduct)
                ⊞ OfferedProductWithTV(pack_nettv_budget)
        ⊞ provides(pack_nettv_budget, watchTVchannels)
        ⊞ ChannelPackage(sport_package)
  ⊞ [?x -> super_package] (Rule: possible_product_option)
```

Query atom
Solutions
Atoms that led to the solution “sport_package”
Name of rule that led to the solution “sport_package”
The subclass axiom that led to “pack_nettv_budget” being a member of the class OfferedProduct

Fig. 2. SWRL declarative debugger



```
◆ the_order
□ Bundle
□ chosen_installations
□ chosen_options
□ chosen_resources
□ for_customer
□ has_possible_hardware_options
□ has_possible_installation_options
□ has_possible_product_options
    ◆ Classic Movies
    ◆ Movie Package
    ◆ Movie/Sport Package
    ◆ PC Security
    ◆ Sport Package
    ◆ Super Package
□ possible_options
```

```
◆ the_order □ has_possible_product_options ◆ Sport Package
                because
     → possible_product_option
◆ ADSL Budget + TV □ has_options ◆ Sport Package
                because
     → each-tv-offer-has-channel-package-as-option
◆ ADSL Budget + TV  is a  ● Offer
                because
     Bundle is a subclass of Offer.
◆ ADSL Budget + TV  is a  ● Bundle
```

Fig. 3. ODASE Rules Workbench

As part of the ODASE toolset we have also developed a “rules workbench” standalone application. This tool allows users to examine the reasons why a particular value was produced (or not produced). It uses the same declarative debugging principles described above, but in a more user-friendly skin.

Instead of exploring a proof tree of the rule evaluation, users interact with an “explanation wizard”, which explains why a particular value was produced without resorting to too many technical terms.

Fig. 3 shows how the fact that “Sport package” is offered as a possible product option for the order is explained in the workbench.

The workbench also includes a “missing answer” explanation tool, which can be used to diagnose why a particular answer was not inferred.

A video demo of the tool can be found at the following address: `http://demo.missioncriticalit.com/odase/rules-workbench/en/index.html`.

7 Benchmarks

In this section we present some benchmarks of the backward-chaining rules engine used in the demo. We show results for our engine compiled to C and Java. We also show results for Pellet 2.0.0-rc7 [4]. Note that all engines give the same results for the benchmark queries. The results are presented in Table 1.

Queries	**ODASE(C)**	**ODASE(Java)**	**Pellet**
Query 1	7.6	25.5	819.5
Query 2	0.8	2.7	0.9
Query 3	0.0	0.2	0.7
Query 4	6.8	19.9	0.6
Query 5	43.6	70.3	0.7
Query 6	3.2	12.1	0.7
Query 1	6.3	25.7	1023.7
Query 2	1.0	3.0	0.9
Query 3	0.1	0.5	2.3
Query 4	5.9	18.4	0.6
Query 5	23.5	68.1	0.5
Query 6	3.2	11.2	0.5
Total	102.0	257.6	1851.6

Table 1. Benchmark results (times in milliseconds)

First we load the demo ontologies and some test input data. The input data corresponds to the values of the input fields in the demo application. Next we run 6 queries. These queries are the same ones that are used to generate the output in the demo. After running the 6 queries we change the input data slightly by selecting a different product (the ”Product Selection” field in the demo). We then run the 6 queries again. Pellet and the Java version of our engine were run under a Nailgun server and we “warmed up” the JVM with a mock run. The machine used for the benchmarks was a 2 GHz Intel Core 2 Duo with 2 GB of memory. Times are in milliseconds.

The difference in performance characteristics between the ODASE engine and Pellet highlights the different design goals of the two engines. Pellet is geared more towards semantic web applications where you want to reason about and explore mostly static data. The ODASE engine, on the other hand, is geared towards using SWRL and OWL for online business applications where data is constantly changing. Pellet is doing a lot of processing up-front, while the ODASE engine is only doing the processing it needs to in order to answer each query.

8 Conclusion

We have used SWRL in several commercial rule-based applications and found that it has several advantages over other non-declarative alternatives:

- No side-effects means no need to prioritize rules or have knowledge of the execution algorithm, simplifying rule design and maintenance.
- True separation of business and IT knowledge.
- Evaluation strategy can be optimized without changing rules.
- Encoded business knowledge can be used in multiple ways.
- Declarative debugging possible.
- Tight and natural integration with modelling language (OWL).
- Easy to extend with domain-specific builtins.

We have also developed a practical backward-chaining rule engine for executing SWRL rules and have found its performance and scalability to be good for real-world business applications.

References

1. 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.
2. P. F. Patel-Schneider, P. Hayes, and I. Horrocks. OWL web ontology language semantics and abstract syntax. W3C Recomendation 10 February 2004.
3. E. Y. Shapiro. *Algorithmic Program Debugging.* The MIT Press, Cambridge, Mass., 1983.
4. E. Sirin, B. Parsia, B. Grau, A. Kalyanpur, and Y. Katz. Pellet: A practical OWL-DL reasoner. *Web Semantics: Science, Services and Agents on the World Wide Web*, 5(2):51–53, June 2007.
5. 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.
6. H. Tamaki and T. Sato. OLD resolution with tabulation. In *Proceedings of ICLP '86*, pages 84–98, July 1986.
7. M. Uschold. Ontology-driven information systems: Past, present and future. *Formal Ontology in Information Systems*, 2008.
8. M. Vanden Bossche, P. Ross, I. MacLarty, B. Van Nuffelen, and N. Pelov. Ontology driven software engineering for real life applications. In *Proceedings of the 3rd International Workshop on Semantic Web Enabled Software Engineering*, June 2007.