OTAGen: A tunable ontology generator for benchmarking ontology-based agent collaboration

F. Ongenae, S. Verstichel, F. De Turck, T. Dhaene, B. Dhoedt, P. Demeester
Ghent University - IBBT
Department of Information Technology
G. Crommenlaan 8/201, 9050 Ghent, Belgium
tel.: +32 9 331 4900 fax: +32 9 331 4899
Email: Femke.Ongenae@intec.UGent.be, Stijn.Verstichel@intec.UGent.be

Abstract

On the one hand, agent-based software platforms are commonly used these days, while on the other hand Semantic Web technologies are also maturing. It is obvious that the combination of these two technologies can bring added value and create a Semantic Agent-based framework. However, it also known that these Semantic Web technologies, and the reasoning on ontologies in particular, can rapidly become resource intensive. In order to get a clear view on this problem, we have developed OTAGen, a highly tunable tool to generate ontologies and corresponding queries. The generated ontologies can then be used to benchmark Semantic Applications, and to define a suitable size and complexity so that the agents can still handle the model.

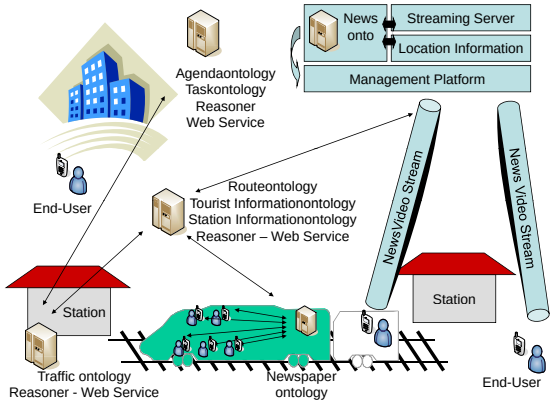

Figure 1. Example Ontology-based Agent platform

1. Introduction

With the constant development and research in Semantic Web technologies, its use in everyday software engineering is increasing. Not least because of newly developed technologies, but certainly because of the improved performance of the tools as well. This also results in an increasing amount of application domains where these technologies are used. One of these new domains, is the support of Context-Aware Services. An example of such a service is the office location service, presented in [7].

Arguably the most important concept in the Semantic Web is the ontology. It describes in a formal and well-defined way the concepts and relationships between these concepts in a particular system, often called the domain. To do so, a machine-processable common vocabulary is used. One of the languages used to describe such an ontology is the Ontology Web Language, OWL. A detailed presentation on the characteristics and functionalty of OWL can be found in [5].

Apart from the fact that OWL is a well defined vocabulary for describing a domain, because of its foundation in Description Logics [1], the model described in this formalism can be used as input for a reasoner to check consistency of the model and infer extra knowledge out of the model. This reasoning process is however a resource intensive and sometimes time-consuming task. It is therefore of paramount importance to have a good idea of the behaviour of the reasoning process in function of the OWL constructs used in the underlying ontology. In the use case presented later in this Section, we envisage to have a network of collaborating Semantic agents. To guarantee the proper functioning of this platform, we need to be sure that each individual agent in the framework remains responsive to incoming requests. It is this requirement that formed the baseline of the research activity presented in this paper.

OTAGen, as the application has been named, was created to support our research in Semantic Agent collabora-

tion frameworks. It is envisaged that in such frameworks, the agents interally use ontologies in two ways. One way is to store the data and information needed for exposing their functionality to clients in a formal ontology-based manner. Also the communication with the clients is in this way formally defined. The other way is, by making use of the fact that ontologies in general and OWL-DL in particular are based on formal first-order logics, to internally reason on the ontology models and infer extra knowledge out of the asserted information. This inferred knowledge is then used to create more intelligent agents.

An example of such a platform is illustrated in Figure 1. A number of different agents can be seen in this diagram. Each of them provide different functionality and data, e.g. tourist information, traffic updates, personal agenda and todo-list information. But not only pure ontology-based agents can be included. The ontology can also be used to manage other agents, such as a media streaming server, to provide the most appropriate video stream for a user with a certain user profile. It is important to note howevever that all these agents internally use the ontology technology, and first-order logic reasoners. Each of these agents provides a well-defined function, which can of course be combined in order to give an even more intelligent service to the end-users.

The main motivation for developing an application able to create different ontologies, with varying complexity and sizes was motivated by the research presented in the previous paragraphs. After all, the main idea behind ontologies is to gather the knowledge, both explicit and implicit, about a certain domain, and make this representation both machine as well as human readable and interpretable. Because the ontology generator now generates random models, it does not contain any domain knowledge, neither any semantics as such. It does however produce fully OWL-DL compliant models of which the characteristics can be tuned and specified.

The use of ontologies as internal representation format, and more specifically the use of reasoners introduce a certain overhead in the processing of that data and information. In order to obtain a clear view on the impact of these new semantic technologies, there is the need to clearly lay-out which characteristics of the ontology greatly influence the performance of the reasoning process. After all, one of the most important performance metrics, the response time of the agent, will be greatly influenced by the time needed to complete the reasoning process. Previous research by the authors [8, 9] has already shown a few indications towards the reasoning bottlenecks.

The remainder of this paper is structured as follows. The following section, Section 2 introduces the ontology technology and related work on benchmarking tools for Semantic applications, while Section 3 details the generating process and its main characteristics. The generated ontologies have been used to evaluate the reasoning process and to find certain characteristics of the ontology which greatly influence the performance of that process. This is presented in Section 4. The last section of this paper presents our conclusions and introduces tracks for future research.

2. Related Work

This sections contains an overview of relevant related work, preceded by a brief introduction to the ontology concept in general.

Although originaly intended for the Semantic Web, ontologies find their application in a variety of areas. Examples of other domains where ontologies have been proved useful are the creation of Location-Based Services or making applications Context-Aware. The use of ontologies to create Context-Aware applications is described in [2] and [6].

Using one of the three sublanguage flavours of OWL, OWL-Lite, OWL-DL and OWL-Full, one can easily adapt to the required expressiveness at hand. Arguably the most interesting sublanguage for many application domains is OWL-DL, balancing great expressiveness with inferential efficiency. The efficiency is guaranteed by the underlying Description Logics [1]. Due to its foundation in Description Logics, OWL-DL is also very flexible and computationally complete.

In the context of ontology generation and benchmarking, three aspects are highly important to remember. Firstly, at the basis of every ontology, there is a T-Box. This is the terminology layer in the ontology, which models the concepts and their relationships, without instantiating these concepts with actual data and information. Secondly, there is of course data, called the A-Box in ontology terms. The last important aspect of a usable ontology are the queries that are commonly executed on the ontology.

Few ontology generators have been developed previously, with the aim to benchmark Semantic Applications and profile their behaviour with different sizes and complexity of the used ontology. Arguably the most well-known is the Lehigh University Benchmark (LUBM)[3]. The LUBM features a university domain ontology, of which the T-Box is statically defined. Also a set of 14 different queries on this data is included in the benchmark. The size of the dataset, the A-Box, can be specified and varied during the generation process of the actual ontologies. The perfomance of Semantic Applications can be measured according to the behaviour of the application, processing the predefined queries on the different generated ontologies.

An extension of LUBM, namely the University Ontology Benchmark (UOB) has been presented in [4]. After all, one of the big disadvantages of LUBM is the statically

Parameter:	Abbreviation:
The seed	seed
The number of classes	nrClasses
The maximal depth of the classes and properties tree	maxDepthTree
The maximal number of direct subclasses	maxNrSubClasses
The maximal number of direct subproperties	maxNrSubProps
The number of logically defined classes	nrLogClasses
The minimal connectivity	minConnectivity
The maximal percentage of object properties or datatype properties with complicated domain/range specifications	maxPercComplDomRange
The number of clusters	clusterCount
The minimum number of classes that have to be included in a cluster	minClusterSize
The maximum number of classes that have to be included in a cluster	maxClusterSize
Specifies the percentage of fully connected instances amongst the instances of the concepts in the cluster	clusterConnectivity
The minimum number of individuals	minNrIndividuals
The percentage of individuals with instantiated datatype properties	PercDatPropForIndividual
The number of times a non-functional datatype property is instantiated	nrDatPropInstances
The minimum number of times a non-functional object property is instantiated (intra-cluster)	minObjPropInstances
The maximum number of times a non-functional object property is instantiated (intra-cluster)	maxObjPropInstances
The number of query depths	nrQueryDepth
The depth of query set number n	queryDepthN
The number of queries generated for set number n	queryCountDepthN

Figure 2. Adjustable parameters of the ontology generation process

defined T-Box. UOB aims to overcome this problem, by introducing OWL-Lite and OWL-DL constructs in the LUBM T-Box, thus creating extra parameters to be varied. The performance of Semantic Applications can again be measured according to the characteristics of the generated ontologies.

However, still a more or less static T-Box is used, although completed with additional OWL-Lite and OWL-DL constructs. Our work with OTAGen aims to provide the capibility of specifying a large range of parameters characterising the ontology, both on T-Box as well as A-Box level. OTAGen also generates queries, of which the characteristics can be specified. All tunable parameters and characteristics of the different generating procedures will be presented in the future sections, more precisely in Section 3,

3. Functional description

In this section the details of the generation process and its characteristics are discussed. An overview of the parameters of the program is shown in Figure 2. The workflow of the generation process is illustrated in Figure 3. It also indicates which parameters are used in each part of the workflow. A deterministic property is added to the generation process by using a seed. Each time a set of parameters is used with the same seed, the same ontology is generated.

A user first specifies the characteristics of the conceptual level (T-Box): the number of (logical) classes, the properties of the subclass tree and the minimum connectivity. A number of properties of the instance level (A-Box) can also be adjusted, namely the number of individuals, datatype and object property instances and clusters. Finally the characteristics of the queries are identified, such as the number of queries and their depth.

As a result the conceptual (T-Box) and instance (A-Box) level of an ontology are randomly and automatically generated. For this ontology also the queries are generated.

3.1 T-Box generator - stage 1

First the necessary simple (not logically defined) classes are created and organized into a hierarchy. Their number is determined by the *nrClasses* minus the *nrLogClasses*. Whether a class has subclasses is chosen at random (chance 1 in 2), but the number of direct subclasses may not exceed the *maxNrSubClasses* parameter. The depth of the trees is also a random choice that doesn't exceed the *maxDepthTree* parameter.

Next the logically defined classes are generated. Their number is determined by the *nrLogClasses* parameter. There are 5 categories that can be created: union, intersection, complement, restriction and enumerated classes. A category is chosen at random. The creation of classes of the last two categories is postponed until the necessary individuals and properties are available. The classes used in the definitions of the other categories are chosen at random from the simple classes.

In the following step the datatype properties are generated for the simple and union classes. The datatype properties of the other categories of logically defined classes need to be computed by a reasoner. The number of datatype properties is a random number dependent on the number of classes, but it is unlikely to have more than 100 datatype properties. The range can be String or Integer. There is also a random chance (1 in 2) that it is a functional property.

Now the object properties can be generated. Each class, simple or union, must have at least *minConnectivity* object properties with other classes (loops do not count).

The minimal number of object properties needed to fulfill this constraint are generated first by using the formula: *([number of union and simple classes] * [minimum connectivity]) / 2.0*. Sometimes an inverse property is generated with a chance of one in five. Each property can get the following characteristics at random: functionality (chance 1 in 3), symmetry (chance 1 in 3) and transitivity (chance 1 in 2).

Properties can have subproperties which is determined at random, with a chance of 1 in 4. The depth of the trees and the number of direct subproperties cannot exceed the parameters *maxDepthTree* and *maxNrSubProps*. The domains of all the properties and the ranges of the object

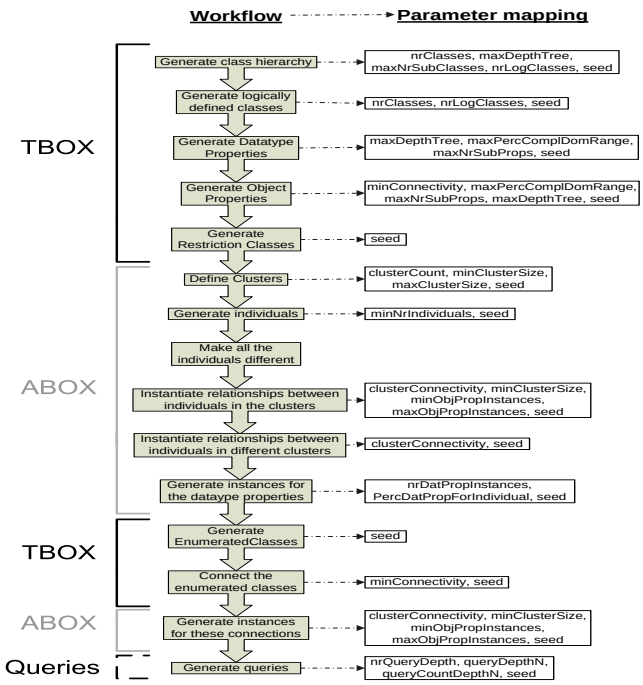

Figure 3. Workflow of the ontology generation process

properties are chosen at random but are dependent on the domain/range of the superproperty if there is one. All the properties can have a complicated domain and the object properties can have a complicated range (consisting of more than one class). This is chosen at random, with a chance of 1 in 4 but the parameter *maxPercComplDomRange* enforces that only a certain percentage of properties can have this characteristic.

Because of the random choice of domain and range, it is possible that all the classes have not reached the minimum connectivity yet. These classes are searched and additional object properties are generated to resolve this issue.

Finally the object properties to connect the future enumerated classes are generated. We do this here so that they can also be used for the generation of the restriction classes. The amount to be generated is the number of enumerated classes multiplied by the minimum connectivity.

In the last step before starting the generation of the instance level, the restriction classes are created. For each restriction class, a random non-transitive, object or datatype property is chosen. There are 5 categories of restrictions: all values from, some values from, maximum cardinality, minimum cardinality and strict cardinality. A category, the cardinality (1,2,3 or 4) for the cardinality restrictions and the range for the value restriction are defined at random.

3.2 A-Box generator - stage 1

The A-Box generating process is also split up into two parts. The first and major part is to generate A-Box individuals for all classes, apart from the enumerated classes as these can only be generated by the T-Box generator after a number of individuals has been created.

At this first stage in the generating process, five subroutines are executed. We start by establishing a number of clusters. Clusters are defined as a number of classes which more strongly relate to one another, then with other classes in the ontology. The number of these clusters to be generated is defined by parameter *clusterCount*. The size of the different clusters in randomly chosen between the values specified by *minClusterSize* and *maxClusterSize*.

The second subroutine generates the actual A-Box instances of the T-Box concepts. The parameter *minIndividualCount* is used as a strict basis for generating a random number of instances of that particular concept, i.e. a random class is chosen of which an individual is generated up until all concepts satisfy the minimum amount as specified by the value of this parameter. Also, the generated individuals are declared to be AllDifferent, with the appropriate OWL axiom.

Once the individuals and the clusters have been defined, the individuals of the concepts belonging to the same cluster have to be linked. As the definition of a cluster indicates, the individuals of the concepts in such clusters are strongly connected. The connectivity of these individuals is specified by the parameter *clusterConnectivity*. The properties of the relationships used to link the individuals together, such as the functionality and symmetry but also the domain and range as specified by the T-Box genertng process, are of course taken into account when inserting this knowledge in the model. The parameters *minObjectPropertyCount* and *maxObjectPropertyCount* are used to calculate between those two values a random amount of relationships to be satisfied in case of a non-functional property.

The same procedure is repeated for linking individuals breaching the borders of the clusters. As of course the connectivity has to be less for these individuals, the complement of the value *clusterConnectivity* is used as value for this property. Also, for non-functional properties the same values are used for the inter-cluster case.

As a last step in this first part of the A-Box generating process, the Datatype properties are defined. Here a parameter is used as well, namely *individualDatatypePropertyPercentage*, to define the amount of individuals of a single class that need to have there Datatype properties filled in.

3.3 T-Box generator - stage 2

An enumerated class is a collection of randomly chosen individuals. The number of individuals chosen to make up an enumerated class depends on the number of individuals in the entire ontology. If there are less than 100 individuals, a random number between 0 and 10 is chosen, else a random number between 0 and 100 is chosen.

The enumerated classes also need to fulfill the minimum connectivity constraints. We already created the necessary object properties for this in a previous step. We just need to fill in their domain and range. Each time a random enumerated class is chosen as domain/range and a random - not necessarily enumerated - class is chosen as range and domain. These properties are also given the classical characteristics at random, being functionality, symmetry and transitivity.

3.4 A-Box generator - stage 2

Once the enumerated classes have been defined using the individuals generated by the previous iteration in the A-Box generating process, extra properties using these enumerated classes as domain or range have been generated. During the second phase of the A-Box generation, the same algorithms are used as described in the first stage for linking the concepts together, but this time only these extra enumerated classes and relationships are taken into account.

3.5 Query generator

As an added value for OTAGen, situated in our overall research concerning collaborating agents, a feature for generting queries has been implemented as well. These queries are generated in the SPARQL[**?**] query language, and the characteristics of these queries can be tuned using the parameter *queryDepth*, which specifies as the name suggests the depth of the query. Also for every query depth, the amount of different queries to be generated can be specified using the parameter *queryCount*.

The algorithm starts by choosing an arbitrary concept in the ontology, and depending on a random number between 1 and 3 either following the ObjectProperty path to another concept or querying for the value of a Datatype property. In the second and third case the sub or superclass property of the class of the individual currently considered is followed. In order to insert extra complexity into the queries, an individual is randomly chosen belonging to the concept where the current querying path has arrived. We can do this because the ontology is actually a graph structure and SPARQL defines graph patterns to be satisfied. Thus by walking through the graph of the ontology, and introducing randomness of values to be queries at each node in the graph, a number of different queries with varying difficulty can be generated.

4. Results

Using the tool described in the previous Section 3, a number of experiments were conducted. We varied the parameters as described in Section 3, each time varying only a single parameter in three different values and storing the ontology on disk, thus creating each time a set of three ontologies.

All measurements were done on Linux Debian machines, running a 2.6.17.14 kernel. These machines each have 512 MB of RAM available and an AMD Athlon(tm) 64 Processor 3000+ processor. The Java version used is 1.5.0.11, with Pellet 1.5.1 as reasoner. We have chosen Pellet as a reference reasoner, because of its use of the tableaux algorithm, commonly used in other reasoners as well. The reasoning metrics considered are the time needed for loading the ontology into the reasoner, validating the loaded ontology, checking consistency of the ontology, classifying the concepts in the T-Box of the ontology, and realizing the individuals in the A-Box according to the classified ontology.

In the following paragraphs, a number of interesting results are described giving an idea about the reasoning bottlenecks. For the development of Semantic Web - based reasoning enabled agent frameworks, these results should be taken into account to tune the performance, e.g. the responsiveness of the agents in the framework.

A number of characteristics appear to have limited influence on the overall reasoning process. For one of the characteristics, being the amount of datatype properties defined per concept, we have measured average values of 16501.6ms, 16692ms and 17053.1ms for the total reasoning time. The values are all average values over a ten-time execution of the reasoning on the same ontology. A standard deviation on the total time needed for the reasoning of 27,34ms, 27,49ms and 40,41ms has been measured on the case of 3, 6 and 9 datatypeproperties per concept resp.

Important measurements for the agent use-case have been obtained when varying the number of individuals. These are plotted in the graph in Figure 4. Here the reasoning metrics are plotted against the ontology where the parameter for the individuals to be created per concept is varied from 10, over 20 to 30 individuals per concept. The amount of concepts is maintained throughout all ontologies. From this diagram it is clear that the realization routine amounts for the largest part in the reasoning process, and also that the amount of individuals greatly influences the time needed for the realization routine. It will have to be kept in mind that increasing sizes of A-Box will result in a drastic perfomance decrease if all these individuals are taken into account during the reasoning process. A stan-

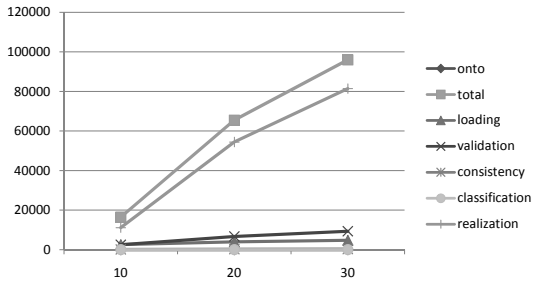

Figure 4. Metrics in ms of the reasoning process in function of the minimum number of individuals per concept

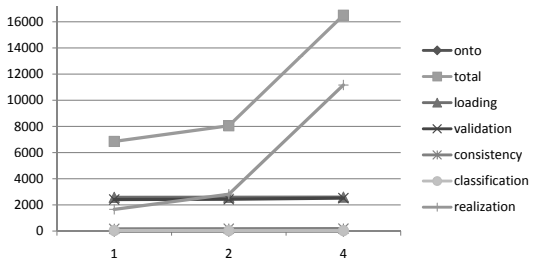

Figure 5. Metrics in ms of the reasoning process in function of the number of logically defined concepts

dard deviation on the total time needed for the reasoning of 27.34ms, 225.04ms and 433.78ms has been measured on the case of minimum 10, 20 and 30 individuals per concept resp.

The last parameter to be described is the amount of logically defined concepts present in the ontology. Again, increasing the amount of such concepts results in heavy increase in the processing time. In the ontologies plotted in the diagram in Figure 5, we have maintained a constant amount of individuals and relationships, but doubled each time the amount of logically defined concepts, going from 1, over 2 to 4. It appears to result in a quadrupled increase in reasoning time. Standard deviations of 2734ms, 20.51ms and 11.57ms have been measured. This result together, with the result from the previous diagram indicates an important feature to be kept in mind when developing Semantic Agent systems. Reassoning over many individuals with a considerable amount of logically defined classes will drastically decrease performance of the agent.

5. Conclusions and Future Work

In this paper we have presented our developments on OTAGen, a highly-tunable ontology generator. An extensive number of characteristics of the ontology can be configured to generate ontologies according to the needs of the user. As indicated as well in the conclusions found in [10], such a generator is needed to get a clear overall picture of the performance of these tools. We have seen that the overall perfomance of Semantic Aware applications and specifically those using OWL and reasoners is greatly influenced by the complexity of the different OWL constructs in the ontology.

Our research and development was motivated by the need to understand the behaviour of these technologies in the context of Agent-Based ontology aware frameworks. This behaviour was measured while varying the tunable characteristics of the ontology. We will therefore take the results into account in our future research, in particular the influence of the A-Box size combined with the number of logically defined classes.

References

[1] F. Donini, M. Lenzerini, D. Nardi, and A. Schaerf. Reasoning in description logics. *Principles of Knowledge Representation*, 1996.

[2] T. Gu, H. Pung, and D. Q. Zhang. Towards an osgi-based infrastructure for context-aware applications in smart homes. *Journal of Pervasive Computing, IEEE*, 3(4):66–74, October 2004.

[3] Y. Guo, Z. Pan, and J. Heflin. Lubm: a benchmark for owl knowledge base systems. *Journal of Web Semantics: Science, Services and Agents on the World Wide Web*, 3(2-3):158–182, October 2005.

[4] L. Ma, Y. Yang, Z. Qiu, G. Xie, Y. Pan, and S. Liu. Towards a complete owl ontology benchmark. *Lecture Notes in Computer Science, The Semantic Web: Research and Applications, Springer Berlin/Heidelberg*, 4011:125–139, June 2006.

[5] D. McGuinnes and F. V. Harmelen. Owl web ontology language overview, February 2004. W3C Recommendation.

[6] D. Preuveneers, J. V. den Bergh, D. Wagelaar, A. Georges, P. Rigole, T. Clerckx, Y. Berbers, K. Coninx, V. Jonkers, and K. D. Bosschere. Towards an extensible context ontology for ambient intelligence. In *Proc. EUSAI 2004, Eindhoven*, pages 148–159, November 2004.

[7] M. Strobbe, J. Hollez, G. D. Jans, O. V. Laere, J. Nelis, F. D. Turck, B. Dhoedt, P. Demeester, N. Janssens, and T. Pollet. Design of casp: an open enabling platform for context aware office and city services. In *Proc. MUCS2007, the 4th International Workshop on Managing Ubiquitous Communications and Services, Munich*, pages 123–142, May 2007.

[8] S. Verstichel, A. Baart, G. Lievens, S. Latre, F. D. Turck, and F. Vermeulen. Train management platform for advanced maintenance of passenger information systems. In *Proc. ITST '07, 7th International Conference on ITS, Sophia Antipolis, France*, pages 1–6, June 2007.

[9] S. Verstichel, M. Strobbe, P. Simoens, F. D. Turck, B. Dhoedt, and P. Demeester. Distributed reasoning for context-aware services through design of an owl metamodel. In *Accepted for publication in the Proc. ICAS '08,*

The Fourth International Conference on Autonomic and Autonomous Systems, Gosier, Guadeloupe, pages 123–142, March 2008.

[10] T. Weithöner, T. Liebig, M. Luther, and S. Böhm. What's wrong with owl benchmarks? In *Proc. of SSWS 2006, The Second International Workshop on Scalable Semantic Web Knowledge Base Systems, Athens, GA, USA*, pages 101–114, November 2006.