Towards a Symptom Ontology for
Semantic Web Applications

Kenneth Baclawski[1], Christopher J. Matheus[2], Mieczyslaw M. Kokar[1],
Jerzy Letkowski[3] and Paul A. Kogut[4]

[1] Northeastern University
[2] Versatile Information Systems, Inc.
[3] Western New England College
[4] Lockheed Martin

Abstract. As the use of Semantic Web ontologies continues to expand there is a growing need for tools that can validate ontological consistency and provide guidance in the correction of detected defects and errors. A number of tools already exist as evidenced by the ten systems participating in the W3C's evaluation of the OWL Test Cases. For the most part, these first generation tools focus on experimental approaches to consistency checking, while minimal attention is paid to how the results will be used or how the systems might interoperate. For this reason very few of these systems produce results in a machine-readable format (for example as OWL annotations) and there is no shared notion across the tools of how to identify and describe what it is that makes a specific ontology or annotation inconsistent. In this paper we propose the development of a Symptom Ontology for the Semantic Web that would serve as a common language for identifying and describing semantic errors and warnings that may be indicative of inconsistencies in ontologies and annotations; we refer to such errors and warnings as *symptoms*. We offer the symptom ontology currently used by the ConsVISor consistency-checking tool, as the starting point for a discussion on the desirable characteristics of such an ontology. Included among these characteristics are 1) a hierarchy of common symptoms, 2) clear associations between specific symptoms and the axioms of the languages they violate and 3) a means for relating individual symptoms back to the specific constructs in the input file(s) through which they were implicated. We conclude with a number of suggestions for future directions of this work including its extension to syntactic symptoms.

1 Introduction

As the Semantic Web initiative and the use of the Web Ontology Language (OWL) [1] continue to develop and grow in popularity, there will be an increasing need for ways to validate ontological consistency and provide guidance in the correction of defects and errors in OWL documents. Even with relatively simple OWL documents, identifying inconsistencies within their XML markup can be a major challenge beyond the capabilities of most writers of ontologies and annotations. Fortunately, formal and heuristic techniques exist for automatically detecting certain types of inconsistencies, and several tools, called "validators" or "consistency checkers", already provide some limited capabilities. The W3C's OWL Test Results page [2] shows nine systems capable of detecting at least some forms of inconsistencies within OWL documents; additional tools with consistency-checking capabilities can be found in [3] and at [4]. As automated consistency-checking techniques continue to mature they will eventually become an integral part of most if not all OWL tools and application development suites. Given the formative stage of these tools, now is the time to consider how they might evolve and explore ways of fostering their ultimate effectiveness and interoperability with other tools.

In this paper we focus on the nature of the output of consistency-checking tools, both in terms of what they are like now and what they might be in the future. In

particular we are concerned with the format and content of the output reports these tools generate, which currently vary widely in the inconsistencies they identify, the identification and description of the detected symptoms of the inconsistencies and the format in which the results are delivered. We begin by reviewing the content and format of the results returned by several existing tools and argue that the lack of a consistent and well-grounded (semantically speaking) approach to the representation of results makes these tools difficult to use, especially by automated programs intended to leverage the results of one or more of them. The situation as it currently exists seems to be begging for the development of a common, shared ontology for describing the symptoms of the inconsistencies discovered in OWL documents. As an example of how such an ontology might work, we present the symptom ontology used by **ConsVISor** [5], the authors' consistency-checking tool, and offer it as an initial step towards the establishment of a symptom ontology for Semantic Web applications. In closing we discuss the strengths and limitations of this nascent ontology along with making a number of suggestions for future improvements and extensions.

2   Review of Existing Consistency Reports

We have analyzed the output from several freely available consistency-checking tools but for this paper we limit attention to the following five: **ConsVISor** [6], **Euler** [7], **FOWL** [8], **Pellet** [9] and **vOWLidator** [10]. The first four of these systems participated in the W3C OWL Test Cases demonstration [2] and either provided links to their full output reports for each test or they were available as Web services from which their results could be easily obtained. The fifth system was included due to its early popularity as a DAML+OIL [11] (the predecessor to OWL) validation tool and because it exhibits some desirable characteristics worth noting. For the sake of comparing the various outputs, we used just one of the W3C's DL inconsistency test cases (DL 109) to produce the sample output reports that appear in Figures 1-5.

Fig. 1. ConsVISor Sample Output Report (condensed HTML version)

In our analysis, we were not as concerned with the correctness of the results as we were with the general nature of the content and format of the reports. On the chosen test case all of the first four systems correctly identified the document as being inconsistent; the fifth system was unable to perform a consistency test due to an error

produced by the parser it happens to use. Of greater interest to us were the answers to three simple questions. Number one, what language does the tool use to format its output? In other words, is the output report written in plain text, structured HTML, OWL/RDF or some non-standard language? The answer to this questions conveys a lot about how easy it is to automatically process the results with other programs. It also identifies whether there is a well-defined semantics for interpreting the results. The second question was, did the tool identify specific symptoms of the inconsistencies within the document? This question is not about whether the tool correctly identified the presence of one or more inconsistencies (i.e., that it responded "Inconsistent" or "Consistent") but rather whether the specific nature of the inconsistencies was identified; just knowing that a document is inconsistent is not as helpful as knowing why. In the ideal case a tool would associate each symptom of an inconsistency with the specific axiom or axioms of the ontology language (i.e., RDF, OWL Lite, OWL DL, OWL Full) that were violated. And the third question was, how useful to the document's author is the output report in identifying and helping to correct the cause(s) of the inconsistency(s)? Being told that a complex document violates a specific OWL axiom will not necessarily provide sufficient information for the author to locate and correct the error, even if she is a highly skilled ontologist. Ideally a tool will, when possible, indicate the line number and character position of the symptom(s) indicative of the underlying inconsistencies.

Fig. 2. Euler Sample Output Report (condensed)

Table 1 provides a summary of the answers to the three questions for the five analyzed systems. The intent of this table is not to pass judgment on particular systems but to indicate the disparate nature of the output of current systems. In the following sections we discuss each of our three questions and their answers in more detail. We also spend time describing the differences between statements made by some of the systems pertaining to *errors*, *warnings*, *information* and *fatal failures*, some of which go beyond the mere identification of symptoms of inconsistencies.

Table 1. Summary of Answers to Three Questions

System	Output Language	Axiom Violations IDed	Symptom Local
ConsVISor	HTML or OWL	yes	yes, if possible
Euler	non-standard	difficult to determine	no
FOWL	non-standard	difficult to determine	no
Pellet	text (html markup)	no	no
vOWLidator	structured HTML	sometimes	no

Test Type: otest:InconsistencyTest
Loading :http://www.w3.org/2002/03owlt/description-logic/inconsistent109.rdf
Adding into XSB:assert(triple('http://www.w3.org/2002/03owlt/description-logic/inconsistent109',rdf_type,owl_Ontology)). assert(triple ('http://oiled.man.example.net/test#Unsatisfiable',rdf_type,owl_Class)). assert(triple(arp46164ARP5354,rdf_type,owl_Class)). assert(triple (arp46164ARP5356,rdf_type,owl_Restriction)). assert(triple(arp46164ARP5356,owl_onProperty,'http://oiled.man.example.net/test#tt')). assert(triple ('http://oiled.man.example.net/test#a',rdf_type,owl_Class)). assert(triple(arp46164ARP5356,owl_allValuesFrom,'http://oiled.man.example.net/test#a')). assert(triple (arp46164ARP5360,rdf_type,owl_Restriction)). assert(triple(arp46164ARP5360,owl_onProperty,'http://oiled.man.example.net/test#tt')). assert(triple (arp46164ARP5360,owl_minCardinality,'3'^^)). assert(triple(arp46164ARP5363,rdf_type,owl_Restriction)). assert(triple (arp46164ARP5363,owl_onProperty,'http://oiled.man.example.net/test#tt')). assert(triple(arp46164ARP5363,owl_maxCardinality,'1'^^)). assert(triple (arp46164ARP5366,rdf_type,owl_Restriction)). assert(triple(arp46164ARP5366,owl_onProperty,'http://oiled.man.example.net/test#tt')). assert(triple (arp46164ARP5366,owl_maxCardinality,'1'^^)). assert(triple(arp46164ARP5365,rdf_first,arp46164ARP5366)). assert(triple(arp46164ARP5365,rdf_rest,rdf_nil)). assert(triple(arp46164ARP5365,rdf_type,rdf_List)). assert(triple(arp46164ARP5362,rdf_first,arp46164ARP5363)). assert(triple (arp46164ARP5362,rdf_rest,arp46164ARP5365)). assert(triple(arp46164ARP5362,rdf_type,rdf_List)). assert(triple (arp46164ARP5359,rdf_first,arp46164ARP5360)). assert(triple(arp46164ARP5359,rdf_rest,arp46164ARP5362)). assert(triple (arp46164ARP5359,rdf_type,rdf_List)). assert(triple(arp46164ARP5355,rdf_first,arp46164ARP5356)). assert(triple (arp46164ARP5355,rdf_rest,arp46164ARP5359)). assert(triple(arp46164ARP5355,rdf_type,rdf_List)). assert(triple (arp46164ARP5354,owl_intersectionOf,arp46164ARP5355)). assert(triple('http://oiled.man.example.net/test#Unsatisfiable',owl_equivalentClass,arp46164ARP5354)). assert(triple('http://oiled.man.example.net/test#a',rdf_type,owl_Class)). assert(triple(arp46164ARP5369,rdf_type,owl_Class)). assert(triple ('http://oiled.man.example.net/test#c',rdf_type,owl_Class)). assert(triple('http://oiled.man.example.net/test#d',rdf_type,owl_Class)). assert(triple (arp46164ARP5372,rdf_first,'http://oiled.man.example.net/test#d')). assert(triple(arp46164ARP5372,rdf_rest,rdf_nil)). assert(triple (arp46164ARP5372,rdf_type,rdf_List)). assert(triple(arp46164ARP5370,rdf_first,'http://oiled.man.example.net/test#c')). assert(triple (arp46164ARP5370,rdf_rest,arp46164ARP5372)). assert(triple(arp46164ARP5370,rdf_type,rdf_List)). assert(triple (arp46164ARP5369,owl_unionOf,arp46164ARP5370)). assert(triple('http://oiled.man.example.net/test#a',rdfs_subClassOf,arp46164ARP5369)). assert(triple ('http://oiled.man.example.net/test#c',rdf_type,owl_Class)). assert(triple(arp46164ARP5375,rdf_type,owl_Class)). assert(triple ('http://oiled.man.example.net/test#d',rdf_type,owl_Class)). assert(triple(arp46164ARP5375,owl_complementOf,'http://oiled.man.example.net/test#d')). assert(triple ('http://oiled.man.example.net/test#c',rdfs_subClassOf,arp46164ARP5375)). assert(triple('http://oiled.man.example.net/test#d',rdf_type,owl_Class)). assert(triple ('http://oiled.man.example.net/test#tt',rdf_type,owl_ObjectProperty)). assert(triple(arp46164ARP5379,rdf_type,'http://oiled.man.example.net/test#Unsatisfiable')).
Total 50 triples
Query : inconsist X Query Success! Result: Inconsist: Class rdf_type Owl_Nothing Exist!

Fig. 3. FOWL Sample Output Report

2.1 Question 1: Output Language

The W3C OWL test results show that in the case of the set of all inconsistency tests no single participating system is currently capable of detecting every inconsistency across all language species (i.e., Full, DL, Lite). On the other hand, every inconsistency test (ignoring one extra credit test) was identified as such by at least one participating system. This result is not very surprising given that the tools employ different techniques for detecting inconsistencies and many of them are still in the early stages of development. This observation, however, does suggest that if we wish to be thorough in the testing of our OWL documents we should run them through more than one of these systems. It is possible to do this today in a manual fashion with those tools that are being considered in this paper, but the process is very tedious at best. Furthermore, the results that are returned by most of these systems are intended for human consumption (and often humans well versed in logic and the interpretation of proof structures); that is to say they are formatted in plain text (or plain text prettified with HTML markup) rather in a language, such as OWL, intended for machine processing and interpretation. It is fair to say that writing a program to automatically run all five systems (or even some reasonable subset) on a given OWL document and then combine the results into a meaningful summary would be challenging. Even if one succeeded in creating such a program it would require tweaking whenever any of the tool authors changed the output representation of their systems or when new, more powerful systems come along, each having its own unique output format.

Results

Input file: http://www.w3.org/2002/03owlt/description-logic/inconsistent109
Level: DL
Consistent: No
Time: 1422 ms (Loading: 1189 Preprocessing: 129 Consistency: 9)

Non OWL-Lite features used
owl:minCardinality restriction with a number greater than 1 3
owl:intersectionOf triples cannot have an anonymous subject in OWL Lite
owl:complementOf construct is used [6ac461:fbdbe20cdf:-7ff4, http://www.w3.org/2002/07/owl#complementOf, http://oiled.man.example.net/test#d]
owl:unionOf construct is used [6ac461:fbdbe20cdf:-7ff7, http://www.w3.org/2002/07/owl#unionOf, 6ac461:fbdbe20cdf:-7ff5]

Fig. 4. Pellet Sample Output Report

If the intent is to be able to have consistency-checking tools that can interoperate with other tools or amongst themselves, then it is paramount that these systems at least provide the option for outputting their results in an ontologically-based language such as OWL/RDF. Once this conclusion is accepted it becomes necessary to consider what should belong in the ontology created for this purpose. Before addressing this issue, however, there are two additional factors we would like to consider regarding the nature of the output of consistency-checking tool.

2.2 Question 2: Axiom Violation

The guidelines provided with the W3C OWL Test Cases specify that a consistency-checking tool "takes a document as input and returns one word being `Consistent`, `Inconsistent`, or `Unknown`" [12]. This approach is fine if all you wish to know is whether an ontology is consistent or not. A Semantic Web browser, for example, may only need this level of feedback from a consistency checker in order to determine whether or not it should attempt to render a document; if the checker returns `Inconsistent` the browser can simply refuse to process it further. If, on the other hand, you are an author needing to produce a consistent document, having a tool that simply tells you your document is `Inconsistent` is not very helpful; rather, you would like to receive some indication of what it is about your document that makes it inconsistent.

Fig. 5. vOWLidator Sample Output Report

So what makes an OWL document inconsistent? An OWL document is inconsistent if there is no interpretation for the RDF graph defined by the document's triples [13]. An interpretation exists if and only if there are no contradictions in the set of triples consisting of the document triples plus all of the derived triples entailed by the axioms of the OWL/RDF language. In theory, if a consistency checker discovers a contradiction in this set of triples it should be possible to trace the contradiction to a specific violation or violations of the OWL/RDF language semantics specified in [13] and [14]. Ideally, a consistency-checking tool should describe the detected symptoms of inconsistencies in terms of the specific language axioms that are violated. By doing so it provides a means for verifying that the tool "understands" what it is talking about. In other words, if a tool can state why it believes there is an inconsistency we have a means for determining whether it is correct. We currently have no systematic way of knowing if the reasoning behind the `Consistent` and `Inconsistent` responses we get from the tools is sound. What

we need is an agreed upon approach for how tools indicate the axiom violations they detect – yet another reason for the creation of a common symptom ontology.

2.3 Question 3: Symptom Location

In addition to identifying the nature of a violation, an ideal consistency checker would indicate the precise location and nature of the "bug" or "bugs" discovered in an OWL document. For example, the tool might tell you that "in the Ontology element you forgot to include an import for the XYZ ontology" or "you mistyped the name of the resource at line 10 character position 5". Unfortunately the precise identification of the underlying cause or causes of an inconsistency, let alone its location, is often not possible. This situation can result from limitations in the methods these tools use to detect inconsistencies, but it can also be due to the fact that it is frequently impossible to determine the original intentions of the author. In either case what these first generation tools usually detect are "symptoms" of problems rather than the problems themselves, and in many cases a single problem can lead to multiple symptoms. It can be very confusing to receive tens of messages from a consistency checker about an undefined resource and yet receive no mention of the construct causing the error itself, such as a typo in the id of the element where the resource was supposedly defined. This is a problem not yet addressed by any existing tool and so for the time being we will need to be content with identifying and locating symptoms rather than bugs.[1]

Some of the tools do a fine job of identifying the nature of the symptoms but few provide much insight into their locations. Again, this is not always possible, but when it is it would be helpful to have this information included in the output report. The question then becomes, where and how is it presented. If we had a common symptom ontology it would be an easy matter to associate a "line" property and a "character-position" property (having appropriate constraints) with the definition of the "symptom" class. Not only would this be useful information to present to human authors today, but in the future, as these tools are embedded in other tools, we can imagine this information being used by a graphical ontology editor to guide the author to the precise location of suspected errors.

2.4 Errors, Warnings, Information & Fatal Failures

All problems are not created equal. In the process of analyzing an OWL document a consistency checking tool may encounter a variety of issues of various degrees of severity. The most egregious are those that clearly violate one or more axioms of the language semantics; these are clearly identifiable as *errors*. In addition to errors, however, there are several types of identifiable "issues" that are not clear violations of the OWL semantics but which none the less indicate that an unintended meaning may be implicit in the document. Consequently, many tools provide *informative* statements that describe, for example, activities such as importing additional ontologies or the making of an assumption about the OWL language class being used (e.g. Full, DL or Lite). *Warnings* represent the identification of constructs (or often, lack there of) that indicate the author may have constructed something other than what was intended, even though it is perfectly valid. For example, it is perfectly permissible in OWL full to use a resource as a property without defining it as such, but it is often the case that doing so represents an oversight in the definition of the property or perhaps a misspelling of the resource name. Informative statements and warnings are not actually indicators of inconsistencies but they are often helpful in

---

[1] The authors are working on a second generation tool, called BugVISor, that attempts to reason from observed symptoms back to the root problems, i.e., the "bugs", that caused them; unfortunately this topic is beyond the scope of the current paper.

providing information that the author can use to ensure the ultimate document conveys the intended meaning. There is finally another type of "issue" not quite captured by any of the other three and that is what we call a *fatal failure*. A fatal failure occurs when the system failed to complete the consistency check because it either crashed, ran into a parsing error or ran out of resources. In such cases the tool (if it can) should return "Unknown" as its result and ideally include as much information as may be useful in determining the cause of the failure.

As with the case of symptom location, there is the question of "where" and "how" to report information about these various types of issues. Once again, having a common ontology would provide the vehicle for representing such information in a consistent and meaningful manner usable by humans and machines alike.

3 Proposed Solution

For the purpose of giving an account of the symptoms of inconsistencies detected in an OWL document by a consistency-checking tool, various approaches are possible. A natural approach, and one chosen by several existing tools, is the use of a plain textual description describing the problem(s) in human readable terms. An obvious extension to this is to beautify the output with HTML markup or to add some structure to the results using, for example, HTML tables. While the latter approach makes it possible to write programs to "screen scrape" the content of the results from the HTML, none of these approaches is well suited to the automated processing and interpretation of the results by machines. As discussed above, there are several advantages to making consistency-checking results machine processible, and, given the intended audience of this paper, we do not believe it is necessary to belabor the argument. Given that the use of a formal language is desirable the question then becomes one of choosing which one based on the requirements of the information that needs to be conveyed. Clearly we will need to define at least a class to represent instances of symptoms and this class will need properties to associate individual symptoms with the specific resources involved, axioms violated, and location information, at the least. These requirements represent a clear case for use of an ontology language and in the context of the Semantic Web the natural choice is of course OWL itself.

If we agree it is worthwhile to develop a common ontology in OWL for use in annotating the results of consistency-checking tools, we can then turn our attention to the question of what it needs to contain. As we have argued, symptoms are about as much as one can expect to receive from the current generation of tools, and even with the advent of more sophisticated systems capable of identifying actual bugs there will always be situations where the best one can do is cite the symptom(s) indicative of a possible bug (this is so because there will always be situations where a system cannot unequivocally determine the intent of the author). For this reason we believe the symptom class should be the focus of the ontology, and have placed it at the center of our proposed high-level design shown in Fig. 6. So what makes up a symptom? First, a symptom is always attributable to a specific OWL document, i.e., the one in which it was detected. In fact, since the output of the tool may actually include multiple symptoms it would be convenient to define a high-level class to describe the OWL document and associate with it the (possibly empty) set of symptoms that were detected. This `Ontology` class could also contain meta-data about the document, such as its OWL level (i.e., Full, DL or Lite), the high-level conclusion reached by the tool (i.e., `Consistent`, `Inconsistent` or `Unknown`) and even whether is was found at the specified location (we return to these ideas in Section 5).

Another characteristic of a symptom is that there should always be (in theory at least) an axiom of the language that was violated and which serves as the justification for claiming the symptom represents an inconsistency. This can be represented using a property on the symptom class that connects it with a class representing axioms. In

addition, a symptom is always manifested by the characteristics of one or more RDF statements (i.e., triples) found in or inferred from the OWL document. It is possible to identify various types of symptom sub-classes based on the statement characteristics they share; when we consider the ontology used by **ConsVISor** in the next section we present an example of a set of symptom subclasses defined in this manner. This idea of creating symptom subclasses based on shared characteristics is appealing because we can then define the relationships between the statements comprising a subclass of symptoms through the definition of appropriate property constraints on the subclass. The examples in the next section will make this more clear, but for now the point is that the ontology needs to be able to associate symptoms with specific statements specified in or inferred from the OWL document, and one convenient way to do this is by defining various symptom subclasses based on shared statement characteristics.
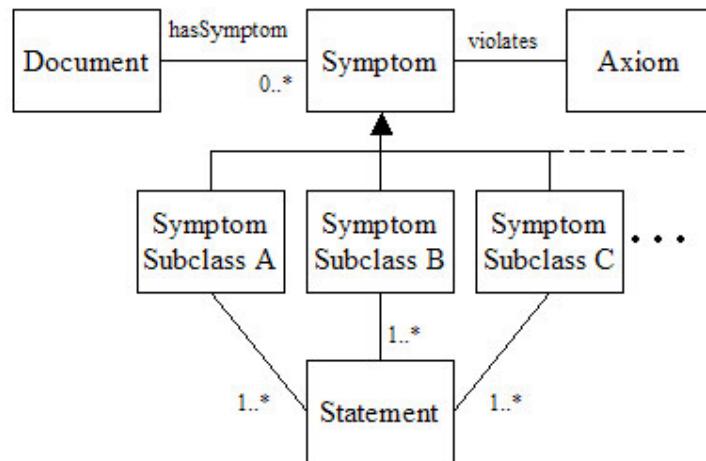

Fig. 6. Proposed High-Level Design for a Shared Symptom Ontology

The UML diagram in Fig. 6 shows a high-level view of the proposed design for a common symptom ontology, as described in the preceding paragraphs. This design is intended to capture the general concepts that we hope can be agreed upon by the Semantic Web user community and thereby serve as a starting point for discussion; consequently, many of the details of a complete solution – including what fully defines the Axiom and `Document` classes – are left unspecified. In the next section we describe the design decisions made during the creation of **ConsVISor**'s symptom ontology and offer the implementation as a case study from which to draw ideas for a community-defined solution.

4   ConsVISor's Symptom Ontology

ConsVISor is a rule-based tool for checking the consistency of OWL Full, OWL DL and OWL Lite documents. At the time of this writing it is freely available for use as a Web service at http://www.vistology.com/**ConsVISor**. **ConsVISor**'s development was initiated as a DARPA funded project in 2002, at which time the target languages were DAML+OIL and RDF and the implementation was done in Prolog and Java. During the conversion from DAML+OIL to OWL in 2003 the system underwent a number of changes including a re-implementation in Jess and Java and the introduction of a symptom ontology for use in producing OWL-annotated results. The success of this ontology-based approach to generating consistency reports led to the conceptualization of this paper. In the rest of this section we describe the design of

the **ConsVISor** symptom ontology and offer it as an initial step towards the ultimate realization of a common symptom ontology for Semantic Web applications.

As stated earlier, a symptom is an indication of a possible problem in an ontology. As with human disease symptoms, such an indication can be benign or it can be severe. The actual diagnosis is not addressed by this ontology. It is only concerned with those conditions that may be of some use in a determination that the ontology has a problem that should be addressed.
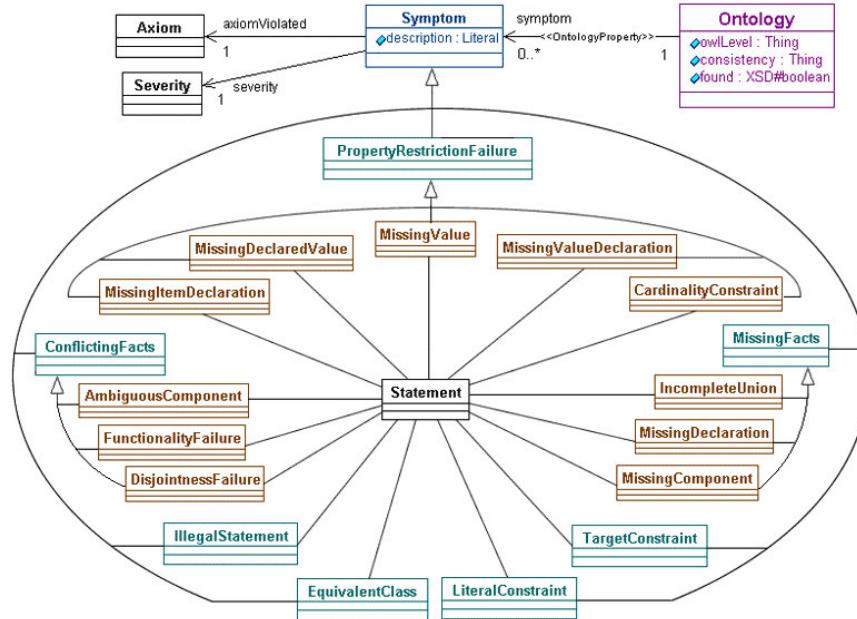

Fig. 7. ConsVISor's Symptom Ontology (simplified for display purposes)

As discussed in the previous section, a particular symptom individual is always characteristic of some particular ontology. Since symptoms are properties of an ontology, they must somehow be linked with the ontology being checked. However, it would be inappropriate to link the symptom directly with the ontology being checked because the symptom is not an intrinsic property of the ontology. It is only a property of the particular invocation of the consistency checker on that ontology. If one regards the symptoms as being in a *report* generated by a tool, then the symptoms should properly reside in that report, and the report (along with many others) can refer to the ontology being checked. Since such a report consists of OWL statements, it is itself an OWL ontology. The instances of the Ontology class in Fig. 7 are these reports, and symptoms in this report are explicitly linked with the report to which they belong by the `symptom ontology` property. By defining the collection of symptoms generated by a consistency checker to be an OWL ontology, it can itself be checked for consistency, as well as processed by any OWL-aware tool. Consistency checking reports have a number of other ontology properties that are used to specify the context within which the consistency checker produced them, such as:

- The consistency checking tool that produced the collection of symptoms is specified by the `checked` ontology property.
- There are three language levels for OWL ontologies. This level is not intrinsic to the ontology, and a single ontology can be regarded as belonging to any one of the three. The `owlLevel` ontology property specifies the level that was used for the consistency checking operation.

- The `consistency` ontology property gives the result of the consistency checking task. There are three possible values:
    1. `Consistent`
    2. `Inconsistent`
    3. `Unknown`
- If the consistency checking tool was unable to download the ontology, then the `found` ontology property is `false`, otherwise it is `true`. This ontology property was introduced to distinguish the case of an ontology that does not exist (usually because its URI was not given correctly) from an ontology that is so complex that the consistency checker cannot make a definitive determination of consistency in a reasonable amount of time. At first a separate symptom class was used for indicating whether an ontology could be downloaded, but such a characteristic is not a feature of the ontology but rather of the mechanism used to download it. For the same reason, it was felt that this ontology property should not be combined with the `consistency` ontology property.
- The `processingError` ontology property is used when the consistency checker fails in its processing either because some internal error (bug) occurred or there were insufficient resources available for the task. Unlike the `found` ontology property, which is usually due to an error by the client, a processing error is entirely the fault of the consistency checker.

The most important part of the Symptom Ontology is its hierarchy of symptom classes. All of these classes are subclasses of `Symptom`, and they share the following properties:

- The `description` of a symptom is an explanation, using ordinary natural language, of the symptom that occurred.
- There are four `severity` levels for symptoms:
    1. Symptoms having no effect on consistency are at the `info` level. These symptoms are simply reporting on entailments that might possibly be indicators of a problem. For example, a resource that was used as the predicate of a statement but was not declared to be a property would generate such a symptom. It is considered to be only informational because the fact that the resource is a property is entailed by its use as a predicate. However, not all entailments result in symptoms. Subsumptions, for example, are too common and routine to merit explicit mention. Currently there is no clear boundary between those entailments that should produce a symptom and those that should not.
    2. A `warning` is a symptom that is not just the result of an entailment, but that also is not a clear indicator of an inconsistency. Ideally, there should not be any of these as they represent a situation in which the consistency checking tool has been unable to decide between consistency and inconsistency. However, every tool has its limitations because consistency checking is computationally very hard in the worst case (and even undecidable for OWL Full).
    3. An `error` symptom is a clear indication of an inconsistency. Such a symptom is definitely a problem.
    4. A `fatal` error means that processing of the ontology did not complete normally.
- The OWL language reference does not have a single listing of all of the axioms of the language. The specification of the three language levels is stated in several ways, some of which do not specify the axioms of all the language constructs. To deal with this complexity, each symptom specifies one or more references to items in the OWL language reference documents that are responsible for the symptom. Each symptom is linked with at most

one member of the `Axiom` class via the `axiomViolated` property. Each such axiom, in turn, is linked to specific places in the OWL language reference documents via the `reference` property. This allows a client to examine the original sources that define the OWL language. Unfortunately, these sources are not formally specified, so these links are only meaningful to a person.
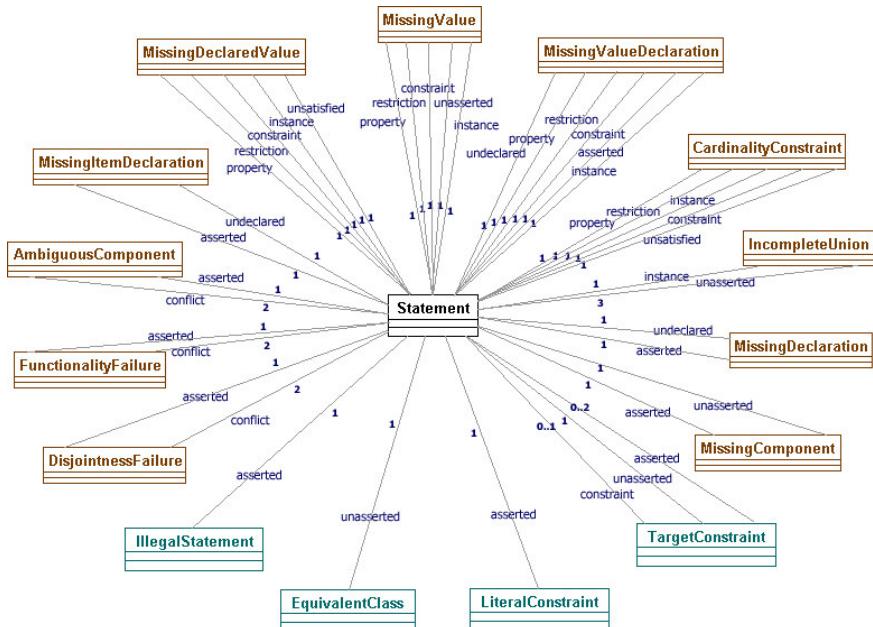

Fig. 8. Property Associations for each Symptom Class

The subclasses of `Symptom` differ from one another with respect to the properties that apply; these differences are depicted in Fig. 8. All of these other properties are alike in linking a symptom to one or more reified statements. These reified statements indicate the statements that were responsible for the symptom. There are several reasons for using reified statements.

- Reified statements are not asserted so they do not represent statements in the consistency checker report. One would certainly expect that the report generated by a consistency checker should itself be consistent so that it can be processed by the same tools that process any other ontology or annotation. If the statements were asserted they would, in general, reproduce the same inconsistencies in the report that exist in the ontology being checked.
- Reified statements are resources, so one can make statements about them. The statements in this case are explanations of the reasons for a symptom being generated.
- One could certainly have explained each symptom by referring to various resources. This was the case in an early version of the Symptom Ontology. However, this design was complex because it required one to introduce a number of auxiliary classes to express all of the concepts. We found that by using reified statements one could eliminate the auxiliary classes. For a while the design used both reified statements and direct references to literals and resources. Eventually the design evolved until all such direct references were eliminated and all explanations were specified using reified statements. The resulting design achieved significant simplifications.

Another design alternative that was considered for some time was for the symptom explanations to be proofs. Some of the symptom classes have sufficient information (including references to axioms in the OWL languages reference documents) to construct a proof. However, we chose not to give complete proof traces because it would make processing of the resulting report much more difficult (both by humans and by software agents). In addition, it would make comparing the reports of different consistency checkers much more difficult if not impossible (even if one could agree on how to compare proofs).

4.2 The Symptom Classes

In this section we define the fifteen symptom classes and identify their properties. The association between the symptom classes and properties are visually depicted in Fig. 8.

AmbiguousComponent. An inverse functional property maps two resources to the same resource. If the two resources are different, then this symptom signals an inconsistency. **Properties**: The `conflict` property gives the conflicting facts, and the `asserted` property gives the inverse functionality constraint assertion.

CardinalityConstraint. A cardinality constraint was not satisfied. This includes max, min and equal cardinality constraints. **Properties**: The cardinality constraint that was asserted is given by the `constraint` property. The property that is being constrained is given by `property`. The resource that is mapped to the wrong number of other resources is given by the `instance` property. The numerical relation (equality or inequality) that failed to hold is given by the `unsatisfied` property. Cardinality constraints are usually specified by asserting a subclass constraint between a class and a relation. This fact is given by the `restriction` property.

ConflictingFacts. Two facts conflict with one another. In other words, they cannot both hold and still be consistent. This is the superclass of several other symptom classes that specify more specific kinds of conflict. **Properties**: The `conflict` property gives the conflicting facts. When there is an explicitly asserted statement that acts as a constraint (such as a functionality constraint on a property) then the assertion is given by the `asserted` property. The difference between the asserted fact and the conflicting facts is that the two conflicting facts are at the same "level" and are similar (such as two resources mapped to the same resource), while the asserted fact is on a different "level" (such as a functionality constraint). If the constraint is built-in then the `asserted` property will not have a value.

DisjointnessFailure. Two disjoint classes have an instance in common. **Properties:** The `conflict` property gives the conflicting facts. When an explicit assertion of disjointness was made, then the `asserted` property gives this statement. Built-in disjointness constraints will not have a value for the `asserted` property.

FunctionalityFailure. A functional property maps a resource to two resources. If the two resources are different, then this symptom signals an inconsistency. **Properties**: the `conflict` property gives the conflicting facts, and the `asserted` property gives the functionality constraint assertion.

IllegalStatement. A statement was asserted that is not allowed at the specified language level. **Properties:** The `asserted` property gives the asserted illegal statement. Sometimes there will also be an `element` item. This occurs when a list of a particular kind (given by the asserted property) contains an illegal element.

IncompleteUnion. A union or enumeration class has an instance that does not occur in the component classes (for a union) or is not one of the enumerators. **Properties:** When there is an explicitly stated union or enumeration constraint, then it is given by the `constraint` property. Built-in constraints are not given. The instance of the union or enumeration is given by the `instance` property. For a union class, the instance should be an instance of one (or more) component classes. These facts may be given using the `unasserted` property. Similarly for enumeration classes.

LiteralConstraint. A literal was asserted to have a datatype with which its value is incompatible. **Properties:** The `asserted` property gives the statement that the literal has the datatype.

MissingComponent. A required resource is missing. For example, `rdf:first` must always have exactly one value. This is a special case of a minimum cardinality constraint, but this symptom class is not a subclass of `CardinalityConstraint`. **Properties:** The `asserted` property gives the statement that uses the resource in question. The `unasserted` property gives the statement that should have been asserted. The object of this statement does not exist, so shown as `sym:unspecifiedEntity`. However, this is just a placeholder. If two symptoms of this kind occur, the unspecified entities need not be the same.

MissingDeclaration. A resource was used in a manner that requires that it be an instance of particular class, but the resource was never explicitly declared to be such an instance. These symptoms are informational only, since the `unasserted` statement is immediately entailed. However, these symptoms are some of the most useful for catching errors. Spelling errors, for example, will result in a `MissingDeclaration` symptom. **Properties:** The `asserted` property gives the statement that uses the resource in question. The `undeclared` property gives the statement that should have been asserted.

MissingDeclaredValue.   This is a combination of `MissingComponent` and `MissingDeclaration`. The value is not only missing, it also must be declared to be an instance of a particular class. This symptom arises from an `owl:someValuesFrom` constraint. This symptom is informational only because the necessary statements are entailed. **Properties:** The `owl:someValuesFrom` constraint is specified using property, restriction and constraint as in the `CardinalityConstraint` symptom. The instance is given by the `instance` property. The missing value is given by `unasserted` as in `MissingComponent`, and the missing declaration is given by `undeclared`.

MissingItemDeclaration. An item in a collection is required to be an instance of a particular class, but it was not declared to be in this class. This is an informational symptom only because the declarations are entailed. **Properties:** The `asserted` property gives the collection-valued statement that constrains the elements of the collection. The particular item that was not declared is given by the `item` property. The declaration that was not asserted is given by the `undeclared` property.

MissingValue. A particular case of an `owl:hasValue` constraint was not satisfied. This is informational only as the statement is entailed. **Properties:** The `owl:hasValue` constraint is specified using property, restriction and constraint as in the `CardinalityConstraint` symptom. The instance is given by the `instance` property. The missing fact is given by `unasserted` property.

MissingValueDeclaration: A particular case of an `owl:allValuesFrom` constraint was not satisfied. This is informational only as the declaration is entailed.

Properties: The `owl:allValuesFrom` constraint is specified using property, restriction and constraint as in the `CardinalityConstraint` symptom. The instance is given by the `instance` property. The statement mapping the instance by the property to an undeclared value is given by the `asserted` property. The missing declaration is given by `undeclared` property.

TargetConstraint: Any of several constraints such as domain and range constraints that have exactly one asserted and exactly one unasserted statement. These are usually informational symptoms, but in some cases the symptom is an error when asserting the unasserted statement is not allowed. **Properties:** The constraint (such as a domain constraint) is given by the `constraint` property. The statement (such as a mapping of a resource that does not satisfy the constraint) that gave rise to the constraint failure is given by the `asserted` property. The statement that should have been asserted is given by the `unasserted` property.

5   Possible Extensions and Enhancements

If a Symptom Ontology for OWL is accepted, then it could be the beginning of a trend toward formalizing the output of many other tools. OWL is most commonly represented using RDF and XML Schema, so that the first step in consistency checking is parsing RDF and XML Schema files (which, in turn, requires parsing XML documents). Both RDF and XML Schema have complex semantics that require their own forms of consistency checking. In fact, RDF has a suite of test cases that is as extensive as the OWL test case suite. [15] The same kind of Symptom Ontology can be developed for RDF as we have done for OWL. Although most of the symptoms would be syntactic, RDF has nontrivial semantics, which, as in the case of OWL ontologies, should be checked for consistency. Both RDF and OWL use XML Schema for the representation of literals, and one can also develop a Symptom ontology for validating XML Schema data types and literals.

More generally, one could formalize the output of compilers for languages other than OWL, RDF and XML Schema. This would allow one to automate the validation of compilers. It would also make it possible to build tools (such as integrated development environments) that process and present the output of whichever compiler one chooses to use.

While developing the Symptom Ontology, we found that there were many axioms that were not expressible in OWL. For example, a `TargetConstraint` symptom has three associated reified statements. In the case of a domain constraint, the subjects of two of the reified statements (the `asserted` and the `unasserted` reified statements) should be the same. One cannot express such a constraint in an OWL ontology. However, it is possible to do so using rules. When an OWL rule language is available, it will be possible to give a more complete theory of symptoms.

6   Conclusion

In their Scientific American article, Tim Berners-Lee, James Hendler and Ora Lassila envisioned the Semantic Web to be "an extension of the current web in which information is given well-defined meaning, better enabling computers and people to work in cooperation" [16]. The success of the Semantic Web depends on tools to ensure that meaning is really "well-defined", i.e., that information is consistent and reflects the intentions of the authors. Many first generation tools have now established that consistency checking of Semantic Web ontologies is feasible. However, for the most part these tools have not themselves adhered to the Semantic Web vision of well-defined meaning and interoperability. Few of them produce

results that can be processed by machine, and there is no shared notion of how to describe the flaws that they detect in Semantic Web ontologies and annotations.

To remedy this situation, we have proposed a common language for identifying and describing semantic errors and warnings that may be indicative of inconsistencies in ontologies and annotations. This language is expressed as an OWL ontology called the Symptom Ontology. This language is currently used by our **ConsVISor** consistency checking tool. Some of the characteristics that we have proposed as being important for such an ontology include those that should be supported by any tool (such as a compiler or interpreter) that requires semantic consistency; namely, a hierarchy of common symptoms and a means of relating the symptoms to the constructs in the source document that gave rise to them. We have also proposed that symptoms should be associated with the axioms of the language that are violated. The latter proposal goes beyond what compilers and other consistency checkers currently do but is essential for achieving the goal of the Semantic Web in which all information is well-defined.

We see the Symptom Ontology as an example of how many tools that currently produce informal, idiosyncratic output can be Semantic Web enabled. Virtually every software tool generates errors and warnings when anomalous situations arise. By formalizing these errors and warnings, their meanings will be formally defined and grounded in the standard for the language, thereby contributing to the Semantic Web vision of meaning and interoperability.

References

1   W3C Recommendation, OWL Web Ontology Language Overview, February 2004. http://www.w3.org/TR/owl-features/
2   W3C, OWL Test Results Page, March 2004. http://www.w3.org/2003/08/owl-systems/test-results-out
3   European OntoWeb Consortium, A Survey of Ontology Tools, May 2002. http://ontoweb.aifb.uni-karlsruhe.de/About/Deliverables/D13_v1-0.zip
4   InfoEther & BBN Technologies, SemWebCentral Validation Tools Assessment. http://semwebcentral.org/assessment/report?type=category&category=Validation
5   K. Baclawski, M. Kokar, R. Waldinger and P. Kogut, Consistency Checking of Semantic Web Ontologies. 1st International Semantic Web Conference (ISWC)}, Lecture Notes in Computer Science, LNCS 2342, Springer, pp. 454--459, 2002.
6   Versatile Information Systems, Inc., **ConsVISor**. http://www.vistology.com/consvisor/
7   AGFA, **Euler**. http://www.agfa.com/w3c/euler/
8   UMBC, **FOWL**. http://fowl.sourceforge.net
9   University of Maryland Institute for Advanced Computer Studies, **Pellet**. http://www.mindswap.org/2003/pellet/
10 BBN, **vOWLidator**. http://owl.bbn.com/validator/
11 DAML, DARPA Agent Markup Language. http://www.daml.org/
12 W3C, OWL Test Cases, February 2004. http://www.w3.org/TR/owl-test/
13 W3C Recommendation, OWL Web Ontology Language Semantics and Abstract Syntax, February 2004. http://www.w3.org/TR/owl-semantics/
14 W3C Recommendation, RDF Semantics, February 2004. http://www.w3.org/TR/rdf-mt/
15 W3C Recommendation, RDF Test Cases, February 2004. http://www.w3.org/TR/rdf-testcases/
16 T. Berners-Lee, J. Hendler and O. Lassila, The Semantic Web: A new form of Web content that is meaningful to computers will unleash a revolution of new possibilities. Scientific American, May 2001.