Property Constraint Generation

Tom Briggs
Shippensburg University
tbriggs@cs.ship.edu

Yun Peng
University of Maryland, Baltimore County
ypeng@csee.umbc.edu

ABSTRACT

A large number of OWL ontologies contain unspecified domains and ranges. Generating the missing constraints has the potential for improving the semantics and run-time of reasoners on the resulting ontologies. We present three algorithms to generate the missing constraints based on terminological evidence using the Disjunction, Least-Common Named Subsumer, and Vivification approaches. The Disjunction approach creates the most specific concept descriptions but adds non-determinism to the reasoner and reduces reasoner efficiency. The Least Common Named Subsumer approach tends to over-generalize concepts and discard more information. The Vivification approach with partial absorption creates concept summaries that generalize concepts with common super classes while preserving the specificity of classes that are not common. The three methods are applied to a large set of real-world ontologies and the resulting relationships between the generated and the original domain and range constraints are compared. The effects on reasoner performance due to the removal of disjunctions are also examined. Vivification is shown to generate constraints that maintain much of the original specificity of the existing constraints while improving the generation and reasoning run-time efficiency.

1. INTRODUCTION

Property constraints are an important component of ontologies because they provide valuable type information about the individuals they connect. By defining the types of individuals that are related by a property, constraints also add information about the way a property was intended to be used. Properties without domains exhibit the analog of the closed-world `NULL` concept - either the domain is unknown or it is not applicable in this situation. If the domain is simply unspecified because the ontology is incomplete, then the result is a loss of information in the ontology.

Constraint generation is a technique to infer constraints from the way a property is used in an ontology. Generating the missing constraints can be used for improving the available information in the ontology, or they may be used to make comparisons to existing constraints. This is similar to the problem of type inference in programming languages such as ML ([9, 13]). Constraint generation can be used to enhance existing Semantic Web tasks such as searching, mapping, and reuse. It can also be used as a tool to aid designers by suggesting constraints from an existing definition.

A more thorough description of unconstrained properties and property descriptions is given in Section 2. Property constraint generation and two generation methods is presented in Section 3. Vivification is presented as a constraint generation method in Section 4. Results comparing the types of constraints generated by each algorithm and computation costs are shown in Section 5. A final summary and discussion of future directions is given in Section 7.

2. BACKGROUND

The Semantic Web integrates knowledge representation languages, reasoning services, and remote linking of resources to create a new generation of World Wide Web agents [6]. The Semantic Web is built from ontologies that formally describe a conceptualization [17]. The Web Ontology Language (OWL) has emerged as the standard ontology representation language for the Semantic Web [2]. OWL, like the Description Logic on which it is based, describes an ontology using class definitions, properties, individuals, and role assertions [1], [7]. OWL reasoners, such as Pellet, use class definitions and property assertions between individuals to infer implicit relationships from the given knowledge [15].

Property assertions are important to the definition of the semantics in an ontology. They define roles, which in turn, define a relationship between two individuals. They are also used to construct complex class definitions using role restrictions. The OWL language allows properties to have a domain and range assertion, which define type concepts that are applied to individuals that participate in in a given role. The type assertions are interpreted through the usual OWL semantics of the subsumes operator $\sqsubseteq$. For every role assertion on property P between individuals a and b, $P(a, b)$, then $a \sqsubseteq domain(P)$, and $b \sqsubseteq range(P)$.

2.1 Unconstrained Properties

In OWL, the default domain and range of a property is the special OWL concept called Thing. An unconstrained property does not necessarily add additional information beyond that a given relationship between individuals holds. Simply asserting that an individual is related to another individual through an object property does not immediately add additional types on those individuals. Suppose a knowledge base contains the definition $\mathsf{Parent} \sqsubseteq \mathsf{Person} \sqcap \exists \mathsf{hasChild}.\mathsf{Person}$ and an assertion $\mathsf{hasChild}(A, B)$. Without domain constraints on the property hasChild we cannot conclude that A is a member of the Parent class, because there may be other types of individuals that use hasChild that have not yet been described in this ontology.

As part of this research, we conducted a comprehensive survey of over 7,000 OWL documents and found that over 75% of the properties were defined without property constraints. There are many reasons that a property may not have a constraint. First, the information may not have been known to the ontology developer. Second, the lack of constraints may have been an artifact of the process to generate an ontology from non-semantic data (e.g. [14]). Other reasons include missing information, user error, or incomplete specification of the ontology.

One other cause is a limitation in the 1.0 version of OWL: domain and range constraints cannot be extended or overridden. There is not, at present, a mechanism in OWL to retract, suppress, or redefine the property's incompatible constraints. To maximize the reusability of an ontology the designer may choose to not define constraints on the properties.

There are a number of reasons that a property is not constrained; however there is insufficient information internal to the ontology to describe why a constraint is left off. We cannot look at a property definition in OWL and decide if the omission of a constraint is intentional or incidental.

3. CONSTRAINT GENERATION

OWL property constraints can contain valuable information, but the majority of properties remain unconstrained. Fortunately, in many cases there is sufficient information in the ontology definition that can be used to generate property constraints. However, because an ontology may be incomplete, using introspection to build a set of constraints from the definitions of an existing ontology will be imprecise and possibly incorrect. Therefore, accuracy of the generator depends on the completeness of the ontology.

Generated property constraints will lead to additional type assertions for individuals connected by the property. In certain circumstances, the additional constraints will allow the reasoner to infer new facts that were previously undiscoverable. In other circumstances, the inclusion of generated property constraints may provide direct evidence of facts that would previously require work for the reasoner to infer.

The way a property is used in an ontology can provide evidence about potential domain and range constraints. There are two sources of this information, namely the terminological and assertional evidence of the ontology. The assertional evidence includes the direct and inferred type assertions and

```
1 Class: Man EquivalentTo Person and hasGender male
2 Class: Woman EquivalentTo Person and not (hasGender male)
3 Class: NFLPlayer EquivalentTo Athlete and
4          playsFor some NFL_Team and hasGender male
5
6 ObjectProperty: hasGender
7 ...
```

Figure 1: Sample Ontology

role assertions. The domain and range constraints must be extracted from the set of class membership assertions for each of the individuals in the knowledge base. This is problematic because property assertions on individuals may be missing, the set of concepts may be overlapping leading to ambiguity, and many ontologies do not define individuals. Information stored in the terminological evidence is more useful for constraint generation.

Lemma 1. *Let P be a property in some ontology, and $C_1, C_2, \ldots, C_n$ be defined classes in the ontology which are subclasses of a role-restriction $C_i \sqsubseteq P.D_i$ involving property P, where D_i is the object of a role restriction. The domain of P must subsume $C_1 \sqcup C_2 \sqcup \ldots C_n$. The range of P must subsume the set of objects $D_1 \sqcup D_2 \ldots D_n$.*

Proof. Let δ and ρ be the domain and range of some property P, respectively. Let $C' = C_1 \sqcup C_2 \sqcup \ldots \sqcup C_n$ represent the set of classes which are subsumed by role-restrictions involving property P. Either the C' subsumes the domain for P and $\delta \subseteq C'$, or it is not. Assume $\delta \nsubseteq C'$. This implies that for some concept $C_i \in C'$, $C_i \nsubseteq P.D_i$, that is that a concept is not a subset of its own definition. This implies that $C_i \neq C_i$, which is a contradiction. Therefore, the domain of a property P must subsume the union of the classes which are subclasses of a role-restriction involving P. A similar proof using ρ and D' will show the same results for the range of a property.

Lemma 1 states that the domain for a property must subsume the set of all classes defined in terms of a role restriction on that property. For the example shown in Figure 1, this implies that the domain for `hasGender` is a class that must subsume `Man`, `Woman`, and `NFLPlayer`. Lemma 1 and Open World Semantics do not preclude other classes that are not yet represented from being subsumed by the domain. By momentarily closing the world, this lemma allows us to construct a constraint from the present state of the terminology. The Lemma does not require the generated constraint to be minimal and does not enforce any future restriction on the relationship of the classes involved in the constraint description.

Lemma 2. *When constructing a constraint from the available evidence in an ontology, for any property in that ontology, there is either a single, trivial constraint definition, or there are an exponential number of possible constraints based on the combinations of classes in the ontology and connectives used in the species of OWL.*

Proof. For the first case, for a single, trivial constraint definition, such as `Thing` or `Nothing`, the property's domain

and range either subsumes everything or the constraint is inconsistent. For the second case, a constraint that cannot be applied to the entire ontology, the number of possible domains is determined by the number of combinations of classes that the constraint can be applied to. If there are n classes (including all direct and indirect super-classes) which are defined in terms of the property, with the standard set of connectives: $\langle \sqcup, \sqcap, and \neg \rangle$, then there are 3^n ways to combine the terms to describe the constraint. Neither will all of these combinations be valid, nor will they be semantically unique.

There are three classes that are defined in terms of role-restrictions on property `hasGender` of the ontology shown in Figure 1. Lemma 1 shows that one domain of this property could be `Man` $\sqcup$ `Woman` $\sqcup$ `NFLPlayer`. This is not the only possible domain. Lemma 2 shows that there may be other domain descriptions. For example, another domain could be `Person` $\sqcup$ `NFLPlayer`, which generalizes the descriptions for `Man` $\sqcup$ `Woman`. The domain could also be `Person` $\sqcap$ `NFLPlayer`, or even `Person` $\sqcap \neg$ `NFLPlayer`.

Even if there were an oracle to assess the quality of a given concept description, an exponential number of queries to the that oracle are required in order to find the 'best' (with respect to the oracle) summarizing concept description. The point that is being made by Lemma 2 is that exhaustively checking all of these possible combinations is intractable for large ontologies.

The following methods use Lemma 1 to find the set of classes that must be subsumed by the domain for a given property. Lemma 2 shows that finding the *best* concept description is intractable, so the following methods seek to find a *good* concept description. The criteria for comparing constraint generation will be based on the quality of the generalization and on the performance of the reasoner on the resulting ontology. The quality metric will be assessed by comparing the relationship between the explicit and generated concepts of the constraints.

3.1 Disjunction Method

One method of generating constraints relies on the construction of the Least Common Subsumer (LCS) of the classes that make up the list of defining classes. Because OWL supports the disjunction of concepts, the LCS of a concept description is the disjunction of its subsumers [4]. Thus, a disjunction list of concepts is added to the property's definition as a domain or range constraint. Computing the LCS is fast and efficient. It runs in time that is polynomial to the number of properties and role restrictions defined in the ontology.

The major drawback to this method is that it tends to create long sentences of disjunctions. There are two problems caused by long disjunctive sentences. First, they tend to add very little information to the type assertions of individuals. Second, disjunctions add an element of non-determinism during reasoning. Tableau reasoners resolve the non-determinism by searching through the possible worlds where each term in the disjunction is validated [4]. Despite improvements in performance due to Tableau reasoners, long chains of disjunctions still require significant time and memory to resolve.

3.2 Least Common Named Subsumer

The LCS has the drawback that it most likely describes a concept that is undefined in the knowledge base. The resulting concept is probably one that is unknown or at least unfamiliar with to the author of the ontology. Another approach to generating domains is to select the Least Common Named Subsumer (LCNS) [5]. The LCNS selects the least specific named class that subsumes the entire set of concepts. This is a trade-off of generality and reasoner performance compared to the disjunction method. Computing the LCNS requires repeated subsumption test calls to the reasoner, which makes its run-time depend on the reasoner.

In certain instances the LCNS approach to generating domain and range constraints represents an improvement over the disjunction approach. The LCNS is able to summarize concept descriptions by finding a single named concept that subsumes some set of classes. This single named concept will replace the long disjunctive sentences that were generated using the LCS method. The performance problems created by the non-determinism are avoided using this approach.

The major drawback is that the LCNS may over-generalize a concept, discarding too much information. Suppose the LCS of a concept is generated to be $A \sqcup B$ where $A \sqsubseteq C$ and $B \sqsubseteq D$, and $\{C \sqcap D\} = \emptyset$. The least common *named* subsumer would be the top concept that represents everything. In OWL, the LCNS frequently over-generalizes to the `Thing` concept. As a method of generating constraints this does not add any useful information at all, since the default constraint is already `Thing`.

4. ENHANCED APPROACH

Vivification describes the process of replacing a selected subset of a disjunction with its LCNS [8]. The vivification approach described here summarizes a disjunction into a series of common partial subsumers. This method will generalize portions of a disjunction into a common, direct, named super-class similar to the LCNS method. Dissimilar classes will remain in the disjunction like the disjunction approach.

Unless there is a mechanism to balance the generalizations made by the Vivification Method, it is susceptible to over-generalization like the LCNS method or over-specialization like the LCS. Over-generalization occurs in the LCNS method when two disjoint concepts were included in the disjunction. Over-specialization occurred in the disjunction approach because there is no attempt to summarize concepts that share a common super-class. The Vivification Method described here uses the relationship between a set of sub-classes in the disjunction list and their common super-classes to balance generalization. When there are a number of classes in the disjunction that share a super-class they will be selected for *absorption* into the common super-class. To moderate the generalization process, absorption will be accepted only if there is sufficient evidence to support the absorption. Sufficiency of evidence will be governed by the *absorption criteria*.

Definition 1 (Absorption Criteria)**.** *Let* $0 \leq \beta \leq 1$ *be*

the Absorption Criteria, a constant parameter. Let L be a disjunctive list of concepts to be summarized, and $A \subseteq L$ be a subset of L to be absorbed by concept description D. Let $m = |A|$ and n equal the number of direct subclasses of D. If $m/n \geq \beta$ then the absorption is accepted, it is rejected otherwise.

Definition 2 (Absorption)**.** *Let $L = C_1 \sqcup C_2 \sqcup \ldots C_n$ be a disjunctive sentence and $A \subseteq L$ the classes to be summarized by absorption. Let C be a class, such that every member of A is a direct subclass of C. Then let m be the size of A, the number of classes in L that share C as direct super-class. Let n be the number of direct subclasses of C. If $m/n \geq \beta$ then the absorption criteria is met, and $L' = L - \{A \cup C\}$, or $L' = L$ otherwise.*

Pseudocode for the absorption operation that conditionally rewrites the disjunction list appears in Appendix A, Figure A. The run-time for this operation depends on the time required to compute the set of direct subclasses for a given common parent which can be assumed to be polynomial in the number of classes. The run-time for the rest of the code is linear in the size of the disjunction list. If duplicates are disallowed from the disjunction list then worst-case run-time is bound by the number of classes in the ontology.

Pseudocode for vivification appears in Figure A in the Appendix. The algorithm starts with a list $L = l_1 \sqcup l_2 \sqcup \ldots l_n$. For each $l_i \in L$ the set of direct superclasses is found and a pair of arrays, C and S, are built to map each disjunction member in C with one of its direct super-classes in S. This strategy is useful for dealing with multiple inheritance when performing absorption. A class can participate in the absorption of each of its direct super classes; and unless it is absorbed by each of its super classes it will not fully be absorbed which helps preserve some of the specificity of the original concept description.

4.1 Absorption Criteria

The absorption criteria parameter, $0 \leq \beta \leq 1$, governs the performance of the absorption algorithm. At its most aggressive, $\beta = 0$, the absorption algorithm will accept every term for absorption. This will cause the absorption algorithm to recursively summarize every concept to `Thing`. On the other hand, when $\beta = 1$, then the only time absorption is performed is when the set to be summarized contains every member of the direct subclasses of another class. In this case, the algorithm will perform identical to Cohen's approach [8].

The value of β is a heuristic to control the action of absorption; and as a heuristic is seems reasonable to require a majority of the direct children of a class are present to be selected for summarization. As a hyper-parameter to the algorithm, its selection will have an impact on its results. We found that values in the range of $0.5 \leq \beta \leq 0.75$ gave satisfactory for results for the ontologies in the Swoogle set. However, for specific ontologies, or even specific properties, there may be other values that are better suited for some task.

Example

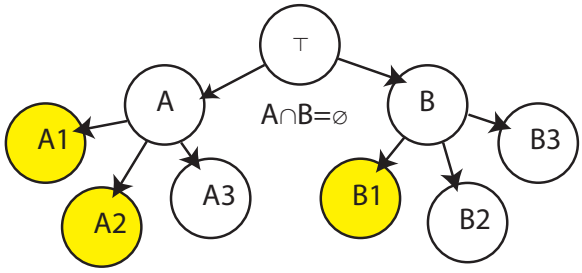

Figure 2: A Class Hierarchy

The ontology shown in Figure 2 consists of a simple hierarchy including two disjoint classes, A and B, each with three direct sub-classes. Let $D = A_1 \cup A_2 \cup B_1$ be the LCS of some concept. Reasoning with the full disjunction on D is unlikely to add any additional information. If the reasoner can conclude that an individual of type D cannot be in class B, then it must be in class A, or vice versa. Similarly, the reasoner must be able to exclude an individual from being in both B_1 and either A_1 or A_2 in order to infer membership in the other class. If exclusionary information is unavailable then reasoning with this disjunction creates unnecessary non-determinism and reasoning complexity.

Because D contains subclasses of A and B then the LCNS of this concept is the top concept ($\top$). The top concept is the only concept that subsumes both parts of D. The LCNS over-generalizes to the top concept which results in this summary adding no additional information. One side-effect is that the non-determinism caused by the disjunction is removed but so is any potential information that may be inferred by the reasoner.

Suppose the absorption criteria is $\beta = 0.5$, then the final vivified summary of $D = A_1 \cup A_2 \cup B_1$ is $D' = A \cup B_1$. The vivified concept generalized over the disjunction of $A_1 \cup A_2$, but preserved the specific B_1 information. The new concept changes the semantics of D, namely an instance of A_3 could now be a instance of D, but it preserves the fact that a member of B_2 could not. Given an instance of D there is still some possibility of inferring membership in A or B_1, but the ability to determine membership in either A_1 or A_2 is lost. Due to this generalization there is still non-determinism in the vivified concept but it is reduced.

5. RESULTS

The three generation algorithms (disjunction, LCNS, and vivification) were executed on a large collection of real-world ontologies in order to determine the effects of constraint generation and to assess the performance characteristics of the algorithms. We will show that a large proportion of real-world ontologies have structures that will add new information through the use of property generation. We will also show that the vivification method produces concept summaries that are nearly of the same quality as the disjunction method and considerably better than the LCNS method. Finally, we will show that the vivification method produces the best reasoning and generation runtime performance.

Table 1: Swoogle OWL Counts

Count	Document Type
2,236,147	Total SWD's including RDF, FOAF, and OWL
133,920	OWL Documents
11,662	OWL SWD's with properties
7,080	Valid OWL Ontologies

5.1 Experiment Design

The Swoogle project is a Semantic Web search engine maintained by the eBiquity group at the University of Maryland, Baltimore County [10]. This experiment uses a snapshot of 2,236,147 semantic web documents (SWDs) including RDFS, OWL, and DAML ontologies, collected by Swoogle between January 2005 and April 2007. OWL documents make up a relatively small proportion of the total collection. The set of documents was further reduced to only those OWL documents that define properties and define classes with property restrictions. The result was a collection of 7,080 documents. Details appear in Table 1.

A property constraint generator, *JPGenerator*, was developed using the Protégé̊API [12]. *JPGenerator* was used to generate domain and ranges using each of the three generation methods and then categorize the relationship between each other and the original property's domain and range. The vivification parameter, $\beta = 0.667$ was used as a fixed constant throughout the experiment.

In order to mitigate run-time dependence on external networked resources, another package, *SwoogleCache*, was developed to use the local copy of SWD's in the collection. The Protégé̊API was modified to use the *SwoogleCache* server as a repository manager for loading the top-level document, and the OWL-API was modified to use the *SwoogleCache* while resolving imported namespaces. These API modifications reduced the time to load documents and eliminated transient errors due to remote servers and network links.

The generator operates by loading an OWL SWD into a Protégé̊model. Imports are resolved and the reasoner is invoked to fully classify the taxonomy and build a complete set of type assertions for all individuals. For each property *JPGenerator* generates the disjunction of classes that are defined in terms of restrictions on that property. In the case of the LCNS and Vivification generation methods the disjunction list is used to compute the summarized concept description. The reasoner is then used to compute the subsumption relationship between the original and each generated concept description.

5.2 Domain Results

The results of the experiment are shown in Table 2. The rows of the table reflect each possible subsumption relationship between the original and generated domain of a property. The result columns show the number times an original domain and its generated domain match the corresponding relationship row. Generally, there three categories for the subsumption relationships: rows 1-6 show a valid generation and subsumption relationship, rows 7 and 8 lacked sufficient information to generate a constraint, and rows 9 and 10 were processing errors.

The first row shows the case where the original and generated constraints were equal. In all cases, the finding that the generated constraint matches the original specified constraint reflects that this algorithm correctly inferred the constraints for the property. The fact that both the LCNS and Vivification methods were able to generate more equivalent constraints than disjunction is a novel outcome. For example, in one real-world ontology, the specified domain for a property `has_pathological_type` is defined by the author as `Diseases` [3]. The disjunction generator found the set of diseases which are defined in terms of a restriction on this property, namely: $\langle$ `Breast_Cancer` $\sqcup$ `Cancers` $\sqcup$ `Adenocarcinoma_of_the_Breast` $\sqcup \ldots$ $\rangle$. Both the LCNS and Vivification algorithms summarized this disjunction to `Diseases`, the same as the original specified domain.

'*Original More Specific Than Generated*' shows a small number of properties where the original was more specific than the generated one. This result was another novel outcome of this approach, it shows cases where the ontology's author incorrectly constricted the constraint beyond those classes which use it. For example, in one ontology the author specified the domain for a property $\texttt{minute} - \texttt{of}$ to be $\texttt{time} - \texttt{point} \sqcap \texttt{calendar} - \texttt{date}$, and the ontology contained the statement $\texttt{calendar} - \texttt{date} \subseteq \texttt{time} - \texttt{point}$ [11]. All three generation algorithms constructed a domain of $\texttt{time} - \texttt{point}$, because that is the only subsumer of the domain. Using the author's definition, an individual filling the property $\texttt{minute} - \texttt{of}$ would be classified as a type of $\texttt{calendar} - \texttt{date}$. Using the generated definition, and individual would be a $\texttt{time} - \texttt{point}$. The former would only be appropriate if the intent of the author was to include a reference to time in a calendar date, whereas the latter would be appropriate if dates referred only to general calendar dates, and not specific instants of time within a particular date (e.g. SQL's DATE vs. TIMESTAMP column types).

The next row, '*Original More General Than Generated*' is a positive to neutral outcome. The present usage of the ontology contains a property where the original asserted constraint is more general than its present usage suggests. There are many reasons for this. This may be intentional: the author elected to leave open future possibilities; or this may be unintentional: the author incorrectly specified the constraint to be overly general. There is insufficient evidence to reliably identify which is the case. If the property were specifically left general to support future work, then the combination of the constraint generation process and some form of default reasoning may help close the semantic gap for the reasoner while leaving open future modifications.

'*Original* $\top$*, Generated* $\top$' is another form of the case that both the generated and specified constraints are equal. In this particular case, they are both equal to `Thing`, which is a special case of equality that was excluded in the first row of this table. This is a neutral result and reflects the incomplete or under-specified nature of some real-world ontologies. One interesting outcome of this case is the demonstration of the tendency of the LCNS to summarize to $\top$. Here, about three hundred more properties were summarized to the top concept. This will happen when the constraint must include

Table 2: Domain Comparison: Original to Generated Types

	Relationship	Disjunction		LCNS		Vivified	
		# props	%	# props	%	# props	%
1	Original Equals Generated	801	2.8	833	2.9	808	2.8
2	Original More Specific Than Generated	7	0.0	7	0.0	63	0.2
3	Original More General Than Generated	141	0.5	103	0.4	74	0.3
4	Original $\top$, Generated $\top$	800	2.8	1111	3.8	807	2.8
5	Original $\top$, Generated More Specific	2427	8.4	2112	7.3	2412	8.4
6	Generated $\top$, Original More Specific	27	0.1	71	0.2	25	0.1
7	Property Unused, Original Specified	3201	11.1	3204	11.1	3190	11.0
8	Property Unused, Domain Unspecified	21385	74.0	21406	74.1	21267	73.6
9	Processor Failed	64	0.2	46	0.2	201	0.7
10	Reasoner Failed	49	0.2	9	0.0	53	0.2
Total		28902		28902		28902	

portions of the inheritance tree which cross branches at the first level. The least-common named subsumer is the top concept. For these approximately three hundred properties ($1111 - 800 = 311$), this is a negative result for LCNS.

'*Original $\top$, Generated More Specific*' shows strong results for the generation algorithm. These are properties which the ontology author used in a restriction, but did not specify a domain constraint. Because the original constraint was unspecified there is little to compare between it and the generated constraint. It is interesting that the numerical difference between the LCNS and disjunction methods is ($2427 - 2112 = 315$), which is further evidence of the tendency of LCNS to over-generalize to the top concept. The vivification approach is almost identical to the disjunction approach in the quality of generalization.

'*Generated $\top$, Original More Specific*' can be a neutral or negative result for all three algorithms. For a given property, the ontology's author described constraints on a property, but the constraint generation algorithm created a constraint of $\top$. In some cases, this may be the result of over-generalization, in others it may be the result of poor ontology design. In one such example, an ontology's author described a universe consisting of `Person` and its various subclasses [16]. The property in question, `hasAunt` is defined by two classes, `Niece` and `Nephew`. These two classes are defined to be equivalent to `Woman` and `Man`, respectively, and `Person` transitively. The domain was specified by the author as `Person`. The domain was generated to be `Person`. The reasoner concluded that the property was equivalent to `Thing`, even though it was originally specified as non-`Thing`.

The next two cases, appearing on lines seven and eight, are neutral results for property generation. These two lines count properties that are defined in an ontology but are not used in any class definitions. In the present state of definition of the Semantic Web, it is clear that the majority of property assertions are made without domain constraints and are not tied to any particular class definitions. This illustrates a clear problem in ascribing any semantic meaning to the property or the individuals connected by it. In a graph-theoretic interpretation of these properties, they represent an arc between a pair of individuals with a label that represents some concept that connects them. It is likely that the ontology author fell to the GENSYM fallacy and assumes that the semantic meaning of the property is derived from the name of the property - it is not [18].

The final two lines of Table 2 represent processing errors. The first represents errors of the generator itself. For example, the generator tried to generate a concept that clashed with the ontology, there was an unspecified programming error, or in some rare cases, ontologies used data type properties as if they were object properties. The second category of errors is generated by the reasoner itself. When invoking the reasoner, the reasoner is allocated a fixed amount of time and memory to perform (one half-hour of time and 1.5 GB of memory). If the reasoner fails to complete within these resources then it is terminated and counted as an error.

5.3 Ranges

Table 3 shows the same type of results as Table 2, but for range constraints on properties. Based on these two tables, the results for range constraints are similar, but not identical, to those for domain constraints. These differences are caused by the fact that the set of subjects for a property generalize differently than the set of objects for a property. For example, consider a fictitious assertion, `knowsStuff`, which may have a domain of `Person` but a range of `Thing`. Unsurprisingly, the generation of a domain will not be a top concept, while the range will be. In spite of the numerical differences, the same qualitative relationships hold between the different categories of generation results.

5.4 Normalized Results

Table 4 provides another view of the results that were shown in Tables 2 and 3. In this table the rows representing unused properties and processing errors are removed from the results. As a result, the benefits of generation become much more evident. All three methods were able to generate constraints that were equal to the originally asserted constraints about 20% of the time. The disjunction and vivification methods produced more specific constraints for properties that were previously unconstrained nearly 60% of the time. The tendency of the LCNS method to over-generalize is even more apparent in this table, happening 26.3% of the time.

As the Semantic Web technologies mature, best-practice design technique will encourage designers to address unused

Table 3: Range Comparison: Original to Generated Types

	Relationship	Disjunction		LCNS		Vivified	
		# props	%	# props	%	# props	%
1	Original Equals Generated	231	0.8	248	0.9	255	0.9
2	Original More Specific Than Generated	6	0.0	6	0.0	17	0.1
3	Original More General Than Generated	172	0.6	147	0.5	138	0.5
4	Original $\top$, Generated $\top$	647	2.2	930	3.2	657	2.3
5	Original $\top$, Generated More Specific	2113	7.3	1839	6.4	2097	7.3
6	Generated $\top$, Original More Specific	361	1.2	392	1.4	365	1.3
7	Property Unused, Original Specified	3403	11.8	3428	11.9	3416	11.8
8	Property Unused, Range Unspecified	21824	75.5	21834	75.5	20959	72.5
9	Processor Failed	102	102	63	0.2	955	3.3
10	Reasoner Failed	43	43	15	0.1	43	0.1
Total		28902		28902		28902	

Table 4: Domain and Range Comparison: Original to Generated Types

Relationship	**Domain**					
	Disjunction		LCNS		Vivified	
	# props	%	# props	%	# props	%
Original Equals Generated	801	19.1	833	19.7	808	19.6
Original More General Than Generated	141	3.4	103	2.4	74	1.8
Original $\top$, Generated $\top$	800	19.1	1111	26.3	807	19.5
Original $\top$, Generated More Specific	2427	57.8	2112	49.9	2414	58.5
Generated $\top$, Original More Specific	27	0.6	71	1.7	25	0.6
Total	**4,196**		**4,230**		**4,128**	

Relationship	**Range**					
	Disjunction		LCNS		Vivified	
	# props	%	# props	%	# props	%
Original Equals Generated	231	6.6	248	7.0	255	7.3
Original More General Than Generated	172	4.9	147	4.1	138	3.9
Original $\top$, Generated $\top$	647	18.4	930	26.2	657	18.7
Original $\top$, Generated More Specific	2113	60.0	1839	51.7	2097	59.7
Generated $\top$, Original More Specific	361	10.2	392	11.0	365	10.4
Total	**3,524**		**3,556**		**3,512**	

Table 5: Normalized Performance

Method	Average	Std. Dev
Disjunction	0.22	1.16
LCNS	0.14	0.14
Vivification	0.08	0.82

properties. As it does then the results will become more like Table 4 than Table 2. This technique will be even more useful in the future as technique to create missing constraints when using ontologies and suggesting constraints when developing ontologies.

6. PERFORMANCE COMPARISON

Table 5 shows run-time performance statistics of each of the generation algorithms. A random sample of 100 ontologies was selected for this analysis. The same ontologies were used for each sample. For each group, the time, in seconds to generate the constraints, and then classify the modified ontology is recorded. The vivification approach is faster than the disjunction at the 92.6% confidence interval, and is faster than the LCNS 76.4% confidence interval.

7. CONCLUSION

Two methods of summarizing the property constraints generated using the disjunction of terms were described: Least-Common Named Subsumer (LCNS) and Vivification. The LCNS builds a concept description of the least-common (or least general) named class in the ontology that subsumes every member of the domain or range. The vivification concept creates summaries by allowing partial subsumption checking while constructing the list.

A comparison of the constraints generated by the three algorithms showed that the LCNS algorithm tended to over-generalize and create constraints that were equivalent to the top-concept or else were more general than the other two methods. The same results showed that the vivification algorithm produces constraints that are closer to the specificity of the disjunction algorithm while still creating useful summaries of common super-classes in the constraint. This is beneficial to preserve as much of the available information as possible while breaking the long disjunctive chains that impacted the reasoner's performance.

A time-based comparison showed that the vivification algorithm was faster for generation and reasoning than the other approaches. This is due to the amortization of the generation costs over the improved reasoner cost of the final model. Thus, for certain ontologies, the vivification algorithm is superior in generalization and in performance.

This work was conducted over a large collection of ontologies using every available ontology in the Swoogle collection. This work includes well engineered ontologies that were clearly generated by experts. It also includes ontologies that were of poor quality containing incomplete semantics and in some cases, errors. In spite of the vast differences in the available semantics each of these methods were able to create property constraints. The quality of the generated constraints will improve as the quality of the ontologies improves.

Another novel outcome of this work was the relationship between the constraints generated through the disjunction and vivification methods. In the majority of instances the two created constraints that had the same subsumption relationship to the original constraints. Even more surprising are the differences in relative cost of generating and reasoning with disjunctive and vivified constraints. The vivification method is clearly the dominate generation method of the three. These results recommend its use as a general purpose constraint generation algorithm.

7.1 Future Work

Two areas for future work include the development a formal approach to selecting the absorption parameter, β, and in demonstrating the utility of using this approach to merge incomplete semantics during a merge operation. The expectation is that the addition of domain and range constraints will help tools overcome the varying degrees of completeness and specificity in source ontologies.

8. REFERENCES

[1] Owl 1.1 web ontology language model-theoretic semantics.

[2] Owl web ontology language reference.

[3] Advanced Computation Laboratory - Cancer Research UK. Cancer ontology. Swoogle ID 4645687.

[4] F. Baader and W. Nutt. *The Description Logic handbook*, chapter Basic Description Logics, pages 41–74, 273. Cambridge University Press, New York, NY, 2003.

[5] F. Baader, B. Sertkaya, and A. yasmin Turhan. Computing the least common subsumer w.r.t. a background terminology. In *Proceedings of Proceedings of the 2004 International Workshop on Description Logics (DL2004). CEUR-WS.org*, pages 400–412. Springer, 2004.

[6] T. Berners-Lee. Semantic web road map. WWW, September 1998. From: http://www.w3.org/DesignIssues/Semantic.html.

[7] R. J. Brachman and H. J. Levesque. The tractability of subsumption in frame-based description languages. In *AAAI*, pages 34–37, 1984.

[8] B. A. Cohen, W. and H. Hirsh. Computing least common subsumers in description logics. *Proceedings of the National Conference on Artificial Intelligence - AAAI*, pages 754–760, 1992.

[9] L. Damas and R. Milner. Principle type-schemes for functional programs. In *POPL '82: Proceedings of the 9th ACM SIGPLAN-SIGACT symposium on Principles of Programming languages*, pages 202–212, 1982.

[10] L. Ding, T. Finin, A. Joshi, R. Pan, R. S. Cost, Y. Peng, P. Reddivari, V. Doshi, and J. Sachs. *Swoogle: a search and metadata engine for the semantic web*, pages 652–659. ACM, New York, NY, USA, 2004.

[11] Knowledge Media Institute, The Open University. Untitled. Swoogle ID 598598.

[12] H. Knublauch. Protègè-api programmers guide. WWW, June 2008.

[13] R. Milner. The standard ML core language. In *ACM Symposium on LISP and Functional Programming*,

1984.

[14] G. Modica. *A framework for automatic ontology generation from autonomous web applications*. PhD thesis, Mississippi State University, 2002.

[15] B. Parsia and E. Sirin. Pellet: An owl dl reasoner. *In Third International Semantic Web Conference - Poster*, page 2003, 2004.

[16] S. U. S. o. M. Stanford Medical Informatics. Family-swrl ontology. Swoogle id 4720353.

[17] R. Studer, V. Benjamins, and D. Fensel. Knowledge engineering: principles and methods. *IEEE Transactions on Data and Knowledge Engineering*, 25:161–197., 1998.

[18] Y. Wilks. *Computational Linguistics and Formal Semantics*, chapter Form and Content in Semantics, pages 257–282. Cambridge University Press, 1992.

APPENDIX
A. PSEUDOCODE

ABSORB(L, A, S, β)

```
1  m ← size[A]
2  B ← FIND-DIRECT-SUBCLASSES(S)
3  n ← size[B]
4  if m ≥ βn
5     then
6          L' = L − (A ∪ S)
7     else L' = L
8  return L'
```

VIVIFY-CONCEPT(L, β)

```
 1  i ← 1
 2  for c ∈ L
 3        do
 4           for s ∈ direct-superclasses[c]
 5                 do
 6                    C[i] ← c
 7                    S[i] ← s
 8                    i ← i + 1
 9
10
11  while done = false
          do
12           A ← SELECT-NEXT-SUBSET(C, S)
13           L' ← ABSORB(L, A, S, β)
14           if L' ≠ L
15              then
16                   DELETESUBSET(A, C, S)
17                   for s ∈ direct-superclasses[S]
                           do
18                            C[i + 1] ← S
19                            S[i + 1] ← s
20                            done ← false
```