

# A Methodology for Asynchronous Multi-User Editing of Semantic Web Ontologies

**Julian Seidenberg**

**Alan Rector**

Bio-Health Informatics Group, School of Computer Science, University of Manchester

jms@cs.manchester.ac.uk, rector@cs.manchester.ac.uk

## ABSTRACT

Current tools, techniques and methodologies for multi-user editing of semantic web ontologies are inadequate. The vast majority of ontologies are maintained by single individuals. However, single user access is increasingly becoming a bottleneck as these ontologies grow in size. We therefore suggest a technique and for locking segments of description logic ontologies for multi-user editing. This technique fits into a methodology for ontology editing in which multiple ontology engineers concurrently lock, extract, modify, error-check and re-merge individual segments of a large ontology. The technique aims to provide a pragmatic compromise between a very restrictive approach that might offer complete error protection but make useful multi-user interactions impossible and a wide-open anything-goes editing paradigm which offers little to no protection.

## 1. INTRODUCTION

Ontologies are the basis of the semantic web. They are the semantic knowledge bases that are fundamental to enabling global interoperability. Providing support for the creation of high quality ontologies is therefore essential for realizing the vision of the semantic web. However, ontology engineering has been a more or less solitary experience. To our knowledge only a very small amount of research currently exists on the topic of multi-user ontology authoring [3] [4]. Such work is essential for the future, since multiple ontology engineers working together as a team can create larger, more complex ontologies, than any single individual working alone. Such large knowledge bases are necessary to drive semantically-aware web applications & services, intelligent business applications and scientific terminologies. We therefore propose a methodology for enabling multi-user ontology editing of a shared ontology.

In the proposed workflow multiple ontology engineers collaboratively edit the same ontology. They do this by placing locks on separate sections of the shared ontology. They then edit these sections independently and asynchronously, merging their changes back into the complete ontology when they are done. Different lock types are used depending on engineers' editing intent. Stronger locks result in more restrictions for other people also working on the ontology.

The key issue for creating a useful methodology is to strike a balance between granting an ontology engineer enough control of the ontology to perform his or her desired editing operations, without locking so much of the shared ontology that other engineers are locked out of sections they want to work on, while also minimizing the number of errors in the concurrent editing environment, as far as possible.

### 1.1 Aim: mitigation of errors

Ghilardi et al. introduce the notion of a conservative extension [7]. A conservative extension is an extension to an existing ontology that does not entail any additional subsumption relations in the base ontology.

Unsatisfiability and subsumption are closely related. Unsatisfiability can be reduced to subsumption and vice versa [13]. Since one common type of error in an ontology is due to an unsatisfiable (inconsistent) concept, locking and editing conservative extensions of shared ontologies would guarantee consistency (and prevent subsumption).

However, our approach *does not aim to ensure that all extensions are conservative*. That is: changes introduced by concurrent users of our locking methodology, may well entail additional subsumption relations (and thereby also potentially cause unsatisfiable classes) for other users of the system. Indeed, we argue that this is a desirable feature, since complete protection would in many cases require extensive locks. Moreover, complete protection would prevent unintended subsumption from taking place (see section 6.4). Indeed, a system of complete protection would be so restrictive that it would defeat the very objective of multi-user ontology

authoring: allowing multiple users to collaboratively construct knowledge structures and serendipitously discover implicit knowledge that results from their interactions (this is elaborated upon in sections 4.4, 6.3.3 and 6.4). We do nevertheless aim to, as far as possible, mitigate common errors [5] that can occur during multi-user ontology editing.

## 1.2 Scope and assumptions

Our multi-user ontology editing methodology is presented within the context of OWL-DL (an ontology language based upon description logics [15]), however it could be easily adapted to other similar formalisms. We assume editing will take place on a single ontology, as opposed to work being carried out on multiple mutually importing modules.

## 2. CONCURRENCY IN DATABASES

The purpose of concurrency control in databases is to allow multiple users to access data in a single database while giving the illusion that each user is alone in accessing the system [1]. In contrast, we defined the aim of multi-user ontology editing as: allowing multiple users to collaboratively construct a knowledge structure. Since the purpose of concurrency control in ontologies is different from concurrency control in database, the solutions developed for database system cannot simply be applied to ontology engineering. Nevertheless, the two problems do have enough in common that a review of the concurrency control in databases will be helpful to set the stage for multi-user ontology editing.

### 2.1 Locking vs. non-locking vs. sagas

Database concurrency control is a very complex topic with many different algorithms. Solutions can be decomposed into locking and non-locking schedulers. Locking schedulers use some variant of the two-phase commit protocol (2PL) to ensure data consistency. Non-locking schedulers use time-stamp ordering or serialization-graph testing to detect data inconsistencies after they occur and roll-back the offending transaction(s) [2].

Sagas are long-running transactions consisting of numerous related individual transactions. If a long transaction were to lock all the parts of the database it required access to, then a large portion of the database would be blocked for the duration of this long transaction. Sagas therefore run in a non-locking fashion, which greatly improves performance. Instead of rolling the entire group of transactions back, if a conflict occurs, sagas run compensating transactions to restore the database to a consistent state. These compensating transactions are application specific and therefore need to be tailored individually for each system [6].

A conservative (locking) approach to database transaction scheduling is best employed when the chance of conflict is high, since it prevents conflicts before they

happen, while aggressive (non-locking) scheduling works best if most transactions complete without conflict, since it eliminates the overhead of locking. Sagas provide optimization performance for long-running transactions.

### 2.2 Database methods applied to ontologies

Campbell [3] points out that systems that initiate some kind of rollback upon detecting a conflict (i.e. non-locking systems) are not good choices for collaborative ontology development, since ontology engineers will have no way of knowing if their work will be discarded when they submit it. The longer the transaction, i.e. the more editing work an ontology engineer has potentially performed before submitting his or her changes back into the main ontology, the greater the potential loss if a conflict occurs. The greater the potential loss, the more reluctant an editor will be to use the system to make his or her changes in the first place.

So, conflict in asynchronous multi-user editing must either be managed by creating compensating transactions (similar to the ones used in sagas), or by locking all parts of the ontology which are to be modified (similar to 2PL). Campbell discusses the former option in detail [3]. His solution requires that the editors discuss every conflict and manually resolve it. This paper suggests a solution using the latter option: preventing conflicts before they occur.

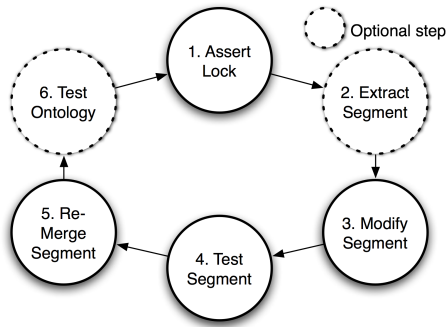
### 2.3 Challenge of locking semantic systems

The additional expressivity of ontologies based upon the OWL description logic over databases creates a new layer of complexity for a system of locks. A database can read, insert and update records. Similarly, axioms in an ontology can be added and removed (but not updated). However, the axioms in an ontology can also reference other axioms, as well as be referenced by existing axioms. Moreover, the nature of these references can vary, so some superclass references might consist of links to named classes, while others might consist of anonymous restrictions (which in turn might contain references to several filler classes). It is because of these differences between ontologies and databases that a novel solution is needed.

## 3. EDITING METHODOLOGY

We have identified a methodology for asynchronous multi-user ontology editing which involves the following steps shown in figure 1.

1. **Asserting locks:** selecting the classes to be modified and specifying the editing intent, i.e. selecting the lock those classes should be locked with.
2. **Extracting locked segment:** extracting locked classes out of the ontology for asynchronous edit. Extracts can either be materialized by using segmentation techniques such as [19] [9] [17], or they can be virtual [14], where the editing application



**Figure 1: The asynchronous multi-user ontology editing workflow**

only allows the modeler to modify classes he or she has locked, but the entire ontology is still present in the background. The former is useful when the ontology is very large and load times become a problem, since segmenting ontologies can significantly reduce ontology size [19] at the expense of potentially relevant classification results (unless the segmentation technique created by Grau et al. [8] is used). The latter allows the modeler to view the entire ontology, allowing him or her to get a better overview of how any changes relate to the global picture, including all classification changes that might be entailed by classes outside the locked segment, as well as allowing the modeler to assert additional locks on-the-fly, if necessary.

3. **Modifying the segment:** the modeler makes changes. During this phase the ontology editing application either records all changes in the background while the modeler is working (as featured in the Swoop ontology editor [11]), or the modified segment is compared to the state of that segment previous to modification (e.g. using the PROMPT suite [16]). Either technique may be used to produce a *change set* that can be applied to or rolled back from the complete ontology as necessary.
4. **Testing the modified segment:** using a description logic (DL) reasoner to find unsatisfiable (inconsistent) classes, running ontology tests and/or running scripted syntactic ontology tests to detect violations in naming conventions, missing annotations and trivially satisfiable classes [20]. The consistency of a modified segment can be checked quickly without computing all new subsumption relations and without building a complete re-merged copy of the knowledge base. This can be done by transforming the series of changes in the modified segmented into one long conjunction and querying a DL-reasoner service as to its consistency with relation to the (presumably) already classified main ontology. The query will not affect the main ontology. Any errors that are uncovered in this process may be further diagnosed by using various ontology debugging tools [10] [20].

5. **Re-merging the modified segment:** a change set is merged back into the main ontology and the corresponding locks are removed. This set is comparable to a *commit* of a long transaction in a conventional database application. The version of the ontology before the re-merge is stored to allow for if a change is found to be erroneous in the future.

6. **Re-test the modified segment:** if the changes were made using an unsafe change mechanism the ontology should again be checked for errors and unexpected inferences at this point. The extra test is necessary because other modelers might have introduced changes since the original segment was taken. Additionally, if the segment was materialized (extracted), additional inferences might result from the additional axiom present in the complete ontology.

This paper will focus on the first and currently least well established part of this workflow: locking ontology segments and thereby mitigating errors.

## 4. RESULTS OF MULTI-USER EDITING

We define a consequence of multi-user ontology editing as any incident that might occur when one user integrates his or her changes into an ontology that has since been changed by someone else. We have identified two basic categories of such incidents: database-type errors well known from conventional concurrency control systems and consequences resulting from semantic inferences made by a description logic reasoner. These incident types may be further broken down as follows:

### 4.1 Incident types

- **Database-like errors:** errors well known from databases and other systems with concurrent multi-user access.
  - **Syntax error:** someone creates syntactically invalid ontology axioms.
  - **Dangling reference:** someone creates a reference to an item that he or she assumes exists, but that in reality has either been deleted, or not yet created. From a purist RDF/OWL viewpoint the missing item would just be created implicitly, but that is rarely the intended behavior in a modeling environment.
  - **Overlapping edit:** two people edit the same thing at the same time. Editors might also concurrently create a class with the same name, but with different restrictions.
- **Consequences of semantic inference:** incidents occurrent from the inference capabilities of description logics and similar semantic technologies. The root cause of these effects can be reduced to basic subsumption, but it is nevertheless useful to separate out the specific cases, since this allows us to distinguish different user intentions.

- **Equivalence:** two classes with different names are inferred to be equivalent when classifying an ontology with changes integrated from another user.
- **Unsatisfiable class (inconsistency):** a class becomes unsatisfiable because of the changes someone else has introduced.
- **Subsumption:** a class is subsumed under an additional new superclass when classifying an ontology with changes integrated from another user.

## 4.2 Incident circumstances

Incidents can be either desirable or undesirable. It is impossible to formally defined whether a particular incident is useful, as this is very much dependent on the judgement of the humans creating the knowledge base. What might be an unwanted side-effect for one party, might be a useful interaction for someone else with a different point of view. For example: consequences of semantic inferences may or may not be desirable, depending on the circumstance. However, database-like errors are almost always undesirable.

- **Intended:** a change has the intended effects.
- **Unintended:** a change that results in unexpected side effects.
- **Failure of intent:** a change that produces a different results of those that were intended.

Table 1 shows the combination of various effects of semantic inference and incident circumstances and gives examples of the resulting scenario.

Incident circumstance	Inconsistency error	Subsumption error	Equivalence error
Intended	successful probe class test	desired inference	desired synonym
Unintended	undesireable conflict (intrinsically linked to subsumption)	serendipitous knowledge discovery	potentially useful knowledge discovery
Failure of intent	probe class test failure	underspecified KB (can detect with probe)	difference in modeler opinion

**Table 1: Incident types and circumstances matrix**

## 4.3 Error recovery strategies

Detecting and removing an error early in the lifecycle offers a significant saving in cost [18]. Naturally, preventing an error before it occurs offers the most substantial cost savings. In a multi-user scenario, finding and correcting an error or undesirable in early testing (stage four, in figure 1) is less costly than finding it in late testing (stage six). This leads us to the following sequence of actions, each progressively less desirable than the previous:

1. **Prevent an error occurring in the first place**

2. **Automatically fix the error as it occurs**
3. **Find and fix the error early in the lifecycle (before remerging changes)**
4. **Find and fix the error late in the lifecycle (after remerging changes)**
5. **Never find the error**

## 4.4 Preservation of useful effects

Contrary to the goal of preventing errors are the following goals of preserving useful multi-user modeling incidents, each progressively less important than the previous:

1. **Enable desired multi-user effects:** probe class tests<sup>1</sup>, distributed subsumption and concurrent addition of synonyms.
2. **Enable serendipitous discovery of knowledge:** unintended subsumptions and some unintended equivalences.
3. **Enable concurrent error checking:** probe class tests and multi-defined term conflicts [3] to highlighting differences in modeler opinion (see section 6.2).

The technique presented in this paper attempts to resolve this dichotomy of goals.

## 5. LOCKS

The technique presented in this paper uses locks to mitigate potential errors. Different types of locks were design for varying degrees of editing freedom. The more editing operations allowed, the greater the amount of the ontology that needs to be locked in order to mitigate errors. Such broader locks will result in other people being more restricted in their editing. Conversely, the less harmful an editing operation is, the less powerful the lock that goes along with needs to be, which, in turn, provides a greater degree of concurrency.

### 5.1 Lock selection

Questioning ontology developers revealed that locking on a class-by-class basis is tedious. A more convenient way of asserting locks is to either have pre-defined modules or selecting single classes to lock and then propagate locks downwards to all subclasses of those classes. Both inferred and asserted subclasses should be included.

<sup>1</sup>Similar to test cases in conventional programming, probe classes are classes which are expected to be inconsistent, equivalent, or subsumed under a specific class. These conditions can be checked after classification by a script. Any variation from the intended result indicates an error / failure of intent.

## 5.2 Lock type overview

Six different types of locks were identified in order to meet the goal of mitigating errors while preserving as high a number of concurrent locks as possible. Each of the following locks is more restrictive than the previous. Locks also build upon another, so a delete-lock includes all the previous lock types.

- **Ghost-lock:** allows a class to be referenced and subclassed but not changed.
- **Parent-lock:** the superclass of any otherwise locked class; such a locked class may be referenced, but not changed.
- **Subclass-lock:** allows a new subclass to be created under an existing class (even if it is parent-locked).
- **Add-lock:** allows any type of new restrictions to be placed on an existing class.
- **Remove-lock:** allows existing restrictions to be removed.
- **Delete-lock:** allows existing classes to be moved or deleted.

## 5.3 Lock propagation

Dynamically asserted modules can be selected by following different types of links, similar to how ontology segmentation by traversal works [19]. A module can be constructed by starting from one target class and selectively following superclasses links, subclasses links, outgoing links to restrictions' filler classes and incoming links from other classes' restriction fillers, or any combination of these.

In our model, those locks which allow modification of a class (i.e. add-, remove- and delete-lock) should automatically also lock all that class' subclasses using the same lock. This allows ontology engineers to quickly select a leaf region of an ontology for editing.

Those locks which prevent others from editing (i.e. ghost- and parent-locks) should automatically recursively lock their superclasses using the same lock. This prevents editing all the way up the root of the hierarchy. Since any change to a class' superclass will affect the original class, all superclasses need to be locked in some way to prevent overlapping edit errors (see section 6.1.3).

Similarly, outgoing and incoming links must be ghost-locked and remove-locked (in the case of a delete-lock on the base class) respectively in order to prevent dangling references (see section 4.1).

A modeler may, in some cases, assert additional locks on a class that is already locked by someone else (for example: adding additional restrictions to a class ghost-locked by someone else).

It is important to note that in our proposed technique locks do not propagate recursively horizontally. That is, referenced and referencing classes might be locked as a result of another lock, but those new locks will not in-turn cause other classes to be locked. Locks do however recursively propagate vertically: upwards to their superclasses (with parent-locks) and downwards to their subclasses (with add-, remove- and delete-locks).

All these interactions are detailed in table 2.

## 6. ERROR MITIGATION

This section gives specific examples of the consequences of multi-user editing (as defined in section 4.1) that are mitigated by the locking system, as well as those that are purposely allowed.

Ideally we would prevent all errors before they occurred and preserve all useful multi-user modeling incidents, but this is not possible. In particular, unintended subsumption and unintended inconsistency are fundamentally intertwined [13]. The mitigation strategy discussed in this section provides a balance between the two extremes. That is: mitigating as many errors as possible, while also preserving as many useful concurrent editing outcomes as possible. It particularly does not attempt to prevent all cases of unintended inconsistency in order to allow useful unintended subsumption to occur.

(Note: diagrams in this section follow the legend given in figure 2.)

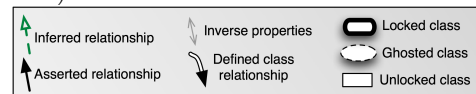


Figure 2: Diagramming conventions

### 6.1 Referential error types

These are basic errors types that have already been solved in other areas such as the databases field.

#### 6.1.1 Syntax error mitigation

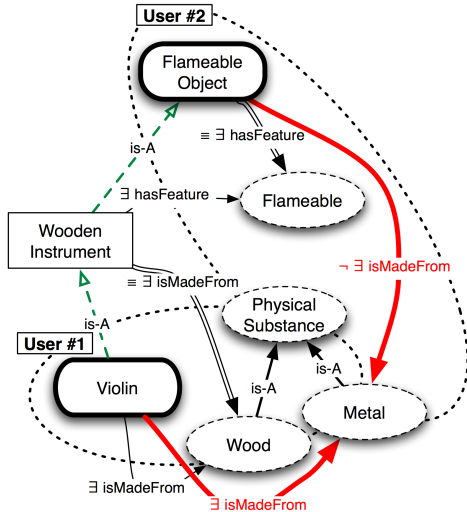
Syntax errors are mitigated by the ontology editor. A good ontology editor will prevent these types of errors. The Protégé-OWL ontology editor [12], for example, makes it impossible to create syntax errors. Protégé checks the syntax of every axiom as it is entered and thereby prevents invalid axioms from being asserted.

#### 6.1.2 Dangling reference mitigation

The locking technique presented herein, which requires that classes be ghosted before being referenced, prevents violations of referential integrity. That is: if a modeler wants to delete a class, it must first be delete-locked. Delete-locking a class requires that all classes referencing that class be remove-lock. Classes that are ghost-locked (being referenced) by other modelers cannot be delete-locked, thereby preventing dangling references.



### 6.3.3 Errors not prevented by locking



**Figure 5: Unsatisfiability not prevented by locking technique**

Figure 5 shows a case of an error that the locking technique, as presented in this paper, cannot prevent. If user #1 adds a restriction to *Violin* that states that it is made from some *Metal* and user #2 simultaneously asserts that *FlameableObject* is not made from some *Metal*, this will result in an inconsistency. The error results because *Violin* is also asserted to be made from *Wood* and everything made from *Wood* is defined as a *WoodenInstrument* and *WoodenInstrument* has a feature that it is *Flameable* and all things that have some *Flameable* feature are defined as a *FlameableObject*. This sequence causes *Violin* to be classified as a subclass of *FlameableObject*, thereby inheriting the restriction that *FlameableObject* is not made from some *Metal*, which contradicts the statement asserted on *Violin*.

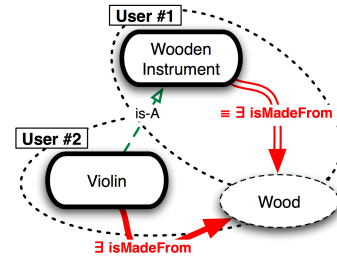
Since the two contradicting statements can be made independently of each other, by separate concurrent modelers, the error will only be noticed late in the lifecycle, when both modelers check-in their changes. The locking technique cannot prevent this (while preserving unintended subsumption).

## 6.4 Unintended subsumption preservation

Preserving the ability for a class to be unexpectedly inferred as a subclass of another class when integrating changes from another modeler is a key feature of our locking technique. However, this same feature makes it impossible to protect from all unexpected inconsistencies (as in the example in section 6.3.3).

### 6.4.1 Common unexpected subsumption

Figure 6 shows a common case of unexpected subsumption. User #1 creates a defined class that asserts that everything made from *Wood* is a *WoodenInstrument*.

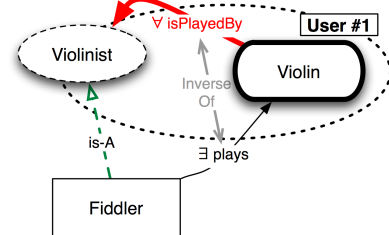


**Figure 6: Common case of useful unexpected subsumption**

User #2 simultaneously makes a new assertion that *Violin* is made from some *Wood*. The result is that *Violin* is inferred to be a *WoodenInstrument*.

This mechanism allows one user to establish definitions and have classes created by other users change location accordingly. The fact that both users are allowed to concurrently ghost-lock the *Wood* class makes this kind of inference result possible. Serendipitous collaborative work can hereby result. The inference mechanism provided by description logic reasoners classifying OWL-DL ontologies provides this potentially very powerful tool for knowledge discovery.

### 6.4.2 Subsumption due to inverse property usage



**Figure 7: Inverse property usage resulting in unexpected subsumption**

Figure 7 shows another unexpected inference that may result during multi-user ontology editing. In the example there is an incoming link from the *Fiddler* class to the *Violin* asserting that *Fiddler* plays some *Violin* and the *plays* property is an inverse property of the *isPlayedBy* property. If a user locks *Violin* and adds a new restriction asserting that *Violin* is played by only *Violinist*, then the logical consequence is that *Fiddler* must be a *Violinist*. The *Fiddler* class was outside the locked section of the *Violin* class since it was not being referenced by it.

All other user of the *Fiddler* class would find it unexpectedly subsumed under *Violinist* when integrating changes from user #1 in this example. The cause for this would not be very obvious. We might additionally imagine difficult to diagnose inconsistencies resulting from contradictory statements made about *Violinist* and *Fiddler*. Nevertheless, we believe the value derivable from unexpected subsumption in a multi-user edit-

ing scenario is greater than the potential harm caused by mysterious inconsistencies.

## 6.5 Unintended failure to subsume

It may be undesirable if, after integrating another modeler's changes, a class is suddenly no longer subsumed under a class it was previously subsumed under. Among other things, removing an item from the conjunction of a defined class, or adding to the restrictions of a primitive class can undo such a previously present subsumption inference. The locking technique again does nothing to prevent such unintended failure of subsumption. However, lack of subsumption can be identified by using the best practice of adding *probe classes* to the ontology (see also footnote 1). Such classes are expected to be unsatisfiable if certain invariants in the ontology hold true. In that way, the *probe class* mechanism can be used to detect and address failures of critical subsumption relations.

Then again, in some cases (such as ones similar to those discussed in section 6.4) an unexpected failure of a subsumption may be the desired correct behaviour. Probe classes are, of course, not necessary in such cases.

## 7. CONCLUSION

We have presented a complete lifecycle for collaborative ontology engineering. This methodology empowers teams of ontology modelers to cooperatively construct the increasingly large and complex knowledge structures. We see users dynamically selecting custom segments for asynchronous editing, instead of relying on a pre-selected module structure. A system of multiple lock types protects ontology engineers from errors as far as possible while preserving useful consequences of multi-user editing.

Existing approaches have either been designed for database-like systems without semantic inferences capabilities [6], do not employ locking to mitigate errors [3], or have been developed primarily to support the safe modular re-use of ontologies [7]. Our approach is distinctive in that it aims to enable asynchronous collaborative construction of rich DL-based semantic-web ontologies while mitigating potential errors only where they do not interfere with serendipitous knowledge discovery.

## 8. REFERENCES

- [1] P. A. Bernstein and N. Goodman. Concurrency control in distributed database systems. *ACM Computing Survey*, 13(2):185–221, 1981.
- [2] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987.
- [3] K. Campbell. *Distributed Development of a Logic-Based Controlled Medical Terminology*. PhD thesis, Stanford University, 1997.
- [4] S. de Coronado, M. W. Haber, N. Sioutos, M. S. Tuttle, and L. W. Wright. NCI Thesaurus: Using Science-based Terminology to Integrate Cancer Research Results. In *MedInfo*, 2004.
- [5] N. Drummond, M. Horridge, H. Wang, J. Rogers, H. Knublauch, R. Stevens, C. Wroe, and A. Rector. Designing User Interfaces to Minimise Common Errors in Ontology Development: the CO-ODE and HyOntUse Projects. In S. J. Cox, editor, *Proceedings of the UK e-Science All Hands Meeting*, September 2004.
- [6] H. Garcia-Molina and K. Salem. Sagas. In U. Dayal and I. L. Traiger, editors, *Proceedings of the Association for Computing Machinery Special Interest Group on Management of Data*, pages 249–259. ACM Press, 1987.
- [7] S. Ghilardi, C. Lutz, and F. Wolter. Did I damage my ontology? a case for conservative extensions in description logics. In P. Doherty, J. Mylopoulos, and C. Welty, editors, *Proceedings of the Tenth International Conference on Principles of Knowledge Representation and Reasoning (KR'06)*, pages 187–197. AAAI Press, 2006.
- [8] B. C. Grau, I. Horrocks, Y. Kazakov, and U. Sattler. Just the right amount: Extracting modules from ontologies. In *Proc. of the 16th International World Wide Web Conference*, 2007.
- [9] B. C. Grau, B. Parsia, E. Sirin, and A. Kalyanpur. Automatic Partitioning of OWL Ontologies Using E-Connections. In *International Workshop on Description Logics*, 2005.
- [10] A. Kalyanpur, B. Parsia, E. Sirin, and B. Cuenca-Grau. Repairing unsatisfiable concepts in OWL ontologies. In *Proc. of European Semantic Web Conference*, 2006.
- [11] A. Kalyanpur, B. Parsia, E. Sirin, B. Cuenca-Grau, and J. Hendler. Swoop - a web ontology editing browser. *Journal of Web Semantics*, 4(1), 2005.
- [12] H. Knublauch, R. W. Ferguson, N. Noy, and M. A. Musen. The Protégé OWL Plugin: An Open Development Environment for Semantic Web Applications. In *Third International Semantic Web Conference (ISWC)*, 2004.
- [13] C. Lutz. Reasoning with concrete domains. In *International Joint Conference on Artificial Intelligence*, pages 90–95, San Francisco, CA, USA, 1999. Morgan Kaufmann Publishers Inc.
- [14] A. Magkanaraki, V. Tannen, V. Christophides, and D. Plexousakis. Viewing the Semantic Web through RVL Lenses. *Journal of Web Semantics*, 1(4):29, October 2004.
- [15] D. L. McGuinness and F. van Harmelen. OWL Web Ontology Language Overview, February 2004. W3C Recommendation.
- [16] N. Noy and M. A. Musen. The PROMPT Suite: Interactive Tools For Ontology Merging And Mapping. *International Journal of Human-Computer Studies*, 59(6):983–1024, 2003.
- [17] N. Noy and M. A. Musen. Specifying ontology views by traversal. In *International Semantic Web Conference*, volume 3298, pages 713–725, 2004.
- [18] W. W. Peng and D. R. Wallace. *Software Error Analysis*. Silicon Press, 1995.
- [19] J. Seidenberg and A. Rector. Web ontology segmentation: Analysis, classification and use. In *15th International World Wide Web Conference*, May 2006.
- [20] H. Wang, M. Horridge, A. Rector, N. Drummond, and J. Seidenberg. Debugging OWL-DL Ontologies: A Heuristic Approach. In *Proceeding of the 4th International Semantic Web Conference*, 2005.