Locking for Concurrent Transactions on Ontologies

Stefan Scheglmann, Steffen Staab, Matthias Thimm and Gerd Gröner

WeST – Institute for Web Science and Technologies
University of Koblenz-Landau
56070 Koblenz, Germany
{schegi, staab, thimm, groener}@uni-koblenz.de

Abstract. Collaborative editing on large-scale ontologies imposes serious demands on concurrent modifications and conflict resolution. In order to enable robust handling of concurrent modifications, we propose a locking-based approach that ensures independent transactions to simultaneously work on an ontology while blocking those transactions that might influence other transactions. In the logical context of ontologies, dependence and independence of transactions do not only rely on the single data items that are modified, but also on the inferences drawn from these items. In order to address this issue, we utilize logical modularization of ontologies and lock the parts of the ontology that share inferential dependencies for an ongoing transaction. We compare and evaluate modularization and the naive approach of locking the whole ontology for each transaction and analyze the trade-off between the time needed for computing locks and the time gained by running transactions concurrently.

1 Introduction

Ontologies, as a prominent knowledge representation approach on the Web, are often collaboratively developed, distributed and extended by multiple users. In general, users modify ontologies independently from each other and they are not aware of edits of other users. Accordingly, approaches for enabling concurrent editing of large ontologies have to ensure that modifications of users are not contradicting each other. Concurrent ontology editing and knowledge base authoring has been the topic of several previous works, which can be roughly partitioned into two categories. First, optimistic versioning-based approaches, like in Karp et al. [10] or ContentCVS by Ruiz et al. [9], make users feel as in a single-user setting — by distinguishing between a private (editable) knowledge base and a public version users can only commit their changes to. In general, commits in these systems consist of multiple changes and these systems provide conflict resolution functionalities. Second, systems like [15] address conflict resolution for parallel editing over a Web interface. The latter systems usually focus more on the social component by making simultaneous changes of different users possible and showing them immediately to all users. In terms of time spans

between two commits/edits of a single user, these two categories are the end-points of a wide spectrum of approaches for dealing with concurrent knowledge base and ontology editing. However, the trade-off between “isolated” access and interleaving operations is also studied in traditional transaction management for databases, which is the foundation for the approach to deal with concurrent access to ontologies. There, sophisticated access methods and protocols avoid unwanted intermediate results and guarantee a consistent synchronization between users. This is achieved by introducing transactions and specific means for handling them. A transaction is defined by an opening statement (‘begin of transaction’), some arbitrary program code that includes interactions with the database and a conclusion statement, i. e., either a ‘commit’ that finalizes the transaction or an ‘abort’ that erases all effects of this transaction. Using a transaction, the individual user should be shielded from influences of other users. The easiest way to achieve such isolation would be a strict serial execution of all transactions. Because individual transactions, however, may contain time-consuming user code, the parallel execution of transactions seems to be a necessity for the performance of the system. Trading off between users’ wishes for isolation from effects of other users led to the notion of serializability [2]. If transactions are scheduled in a—typically interleaved—way that is equivalent to some serial schedule of the same transactions, then the schedule is called *serializable* and the program code defining the transactions behaves functionally as if it has exclusive access to the database. Obviously, such a scheme is not only desirable to have for databases but is also highly desirable to have in the case of frequently accessed ontologies. However, there arise several issues that need to be tackled to carry over the notions of ‘transaction’ and ‘serializability’ from databases to ontologies: (1)The notion of ‘serializability’ is based on the notion of ‘equivalence’ of transaction schedules, but what does it mean that two schedules are equivalent if also computational inferences in ontologies need to be accounted for? (2)As will be shown below, ‘serializability’ is typically based on locking data items such that different transactions do not interact with each other. But what should be locked when logical inference comes into play? (3)Locking data items for transaction scheduling is beneficial as the actual locking process is computationally cheap. However, in the context of ontologies computing the axioms to be locked may become computationally expensive. What is the trade-off between concurrency of ontology access and determining the locks for transactions on an ontology? To illustrate the above challenges, we consider the following example:

Example 1. Let $\mathcal{O} = (\mathcal{T}, \mathcal{A})$ be an ontology with the following axioms in the $\mathcal{T}$-Box:

$$A_1 \equiv \forall R.D_1 \quad (1) \qquad A_2 \equiv \forall R.D_2 \quad (2) \qquad D_1 \sqcap D_2 \sqsubseteq \bot \quad (3)$$

$$B \sqsubseteq D_1 \quad (4) \qquad A \sqsubseteq \forall R.B \quad (5)$$

and some arbitrary ABox $\mathcal{A}$. Assume that one user intends to replace Axiom (4) $B \sqsubseteq D_1$ by a new Axiom (6) $B \sqsubseteq D_2$. Imagine a second user is asking (at the same time) for all concepts that subsume A. Before the change of the first user (replacement of Axiom (4) by (6)), concept A_1 subsumes A, but after the

change, A_2 subsumes A. Before the first user starts the transaction the result to the second user's query would be $A \sqsubseteq A_1$ and afterwards $A \sqsubseteq A_2$. However, the relationship does not hold after the first user has deleted Axiom (4) and not yet added Axiom (6), the result would be neither $A \sqsubseteq A_1$ nor $A \sqsubseteq A_2$.

In this paper, we present a locking-based framework for handling concurrent transactions on ontologies. We define a notion of conflict that prevents different transactions to be executed in an arbitrary way (Sect. 4) and adapt a two-phase locking approach from databases [3] (Sect. 5). Whenever a user issues some operation the necessary locks are acquired. For computing the locking areas for transactions, we utilize the modules of an ontology [6].

2 Foundations and Related Work

The first part of this section introduces fundamentals on concurrent transactions and locking principles, rooted in the database research field. The second part gives an overview on related work of concurrent ontology editing.

2.1 Foundations of Transaction and Locking

Transaction management guarantees the isolation of a transaction execution from the inference with other transactions. Transactions in databases ensure the following properties [2]: *Atomicity:* A transaction is either completely executed or not executed; *Consistency:* The execution of a transaction has to maintain the consistency of a database; *Isolation:* The execution of a set of transactions has the same effect as all transactions would be executed individually; *Durability:* After executing a transaction, all modifications need to be stored in the database. Technically, isolation can be ensured by *serializability*, which guarantees that the outcome of a schedule is equal to the outcome of the same transactions executed one after the other. Such a schedule is called *serializable*. The serializability is guaranteed by concurrency control mechanisms like locking, e. g., the *two-phase locking (2PL)* [3], where data of potential competing transactions are locked in two phases: In the 'expanding phase', the transaction successively tries to acquire locks for the resources of each single *atomic operation*. If it successfully acquires a lock then it performs the operations and continues. If the resource of an operation is already locked by another transaction, the current transaction will stop and consecutively try to acquire a lock for this resource until it succeeds. After all operations of an transaction are performed, the transaction will enter the 'shrinking phase' and free all of its locks.

Following this line of argumentation, a key issue is to determine the resources that need to be locked in order to execute an atomic operation. Obviously, the locked area should be as small as possible to enable interleaving transactions, but the area should be as large as necessary to avoid conflicts.

2.2 Concurrent Ontology Editing

The need for concurrency control in knowledge bases was already acknowledged by Chaudhri et al. [4]. They show the inadequacy of concurrency control mechanism from databases and present Dynamic Directed Graph (DDG), a concurrency control mechanism for rule-based knowledge bases. Their setting and approach is similar to ours but use a very restrictive knowledge representation formalism which simplifies transaction schedule computations.

Other approaches can be roughly partitioned into two categories. The first category [10, 9] extends versioning systems to the knowledge base setting and implement an optimistic conflict resolution schema. The second category [15] applies ideas of online editors to the field of collaborative ontology editing, without considering issues of conflict resolution directly. The rationale behind using these two approaches base on different assumptions. For the first category, it is assumed that knowledge bases are created over a large period of time. For both, it is assumed that the areas of responsibility of different contributors are relatively independent, i. e. they usually modify different parts of the knowledge base. However these assumptions do not necessarily hold in many of application areas, where e. g. already deployed ontologies are modified more frequently. In this paper, we focus on scenarios that need not satisfy these assumptions. Nonetheless, we now look at some of these approaches in more detail.

In [10], Karp et al. introduced an authoring tool for knowledge bases based on frame logic, a predecessor of modern ontology languages. Along with the collaborative subsystem they define the notion of conflicts regarding knowledge base operations and they provide conflict detection mechanisms for the merge process. A similar approach is pursued in ContentCVS [9]. The authors adopted the popular concurrent versioning approach CVS to the field of collaborative ontology development. They include structural and semantic-based conflict detection and state-of-the-art ontology debugging and repair techniques to help the user in conflict resolution. Both approaches make use of an optimistic versioning based approach which detects and resolves conflicting edits in commits on merge time without locking. In [15], Tudorache et al. evaluate the collaboration features of WebProtégé within an intense user study during the development process of the 11th version of the International Classification of Diseases (ICD-11). WebProtégé's collaborative features are all directly integrated in the editing process and make all users aware of all edits currently happening. Additional WebProtégé provides features for incorporating, tracking and reviewing changes on-the-fly. This way of collaborative ontology editing is focusing on conflict prevention or just-in-time conflict resolution. To provide the users of such an editor with useful information about possible conflicts resulting from their edits, an approach similar to our approach could be facilitated. In such a setting our approach would not lock resources but make users aware of possible conflicts calculate from the current edits.

For further related work, Falconer et al. [5] describe patterns of editing behavior and roles of the contributors for large scale ontology-development projects. This is of particular interest for the design and implementation of collabora-

tive editing environments for ontology. The concurrency control mechanism, described in this paper, builds the basis for such systems and the calculation of areas affected by a transaction might benefit from contributor roles and predefined behavior patterns.

For OWL ontologies, Seidenberg and Rector [13] discuss basic principles for multi-user ontology editing. They indicate that due to *inference capabilities* the computation of locking areas goes beyond transaction management principles in databases since changes of a class might lead to different subsumptions of *other* classes, for instance: (i) classes with different names are classified as equal; (ii) a class is classified as a new subclass of a new/changed class; (iii) a class might become unsatisfiable. In this paper, we tackle this indicated challenge of computing locking areas for transaction management.

In order to handle locking, it is necessary to identify areas that are affected by a transaction. Subsequently, we call such areas of an ontology the *area of influence* of a single operation. These areas are obtained by computing modules, either in terms of structural areas, which are built by traversal techniques [14, 11], or in terms of semantic influence areas [6], as it is used in our work.

3 Preliminaries

In this section, we introduce description logics [1], the language family that underlies modern ontology languages like OWL2 [8]. For purpose of presentation, we refer to $\mathcal{ALC}$, but our approach can be generalized to any other description logic where module computation [6] is supported. The signature $Sig_{\mathcal{L}} = \mathbf{C} \uplus \mathbf{R} \uplus \mathbf{I}$ of $\mathcal{L}$ is composed of a set $\mathbf{C}$ of atomic concepts denoted by $A, B, C, \ldots$, a set $\mathbf{R}$ of atomic roles denoted by $r, s, \ldots$, and a set $\mathbf{I}$ of individuals denoted by $a, b, c, \ldots$, and subsets of $Sig_{\mathcal{L}}$ are denoted $\mathbf{S}, \mathbf{S}_1, \mathbf{S}_2, \ldots$. Concepts in $\mathcal{L}$ are built using the symbols in $Sig_{\mathcal{L}}$ and the following syntax rules:

$$C ::= A \mid \top \mid \bot \mid (\neg C) \mid (C \sqcap C) \mid (C \sqcup C) \mid (\exists\, r.C) \mid (\forall\, r.C) \mid$$

where $A \in \mathbf{C}$ is a concept name, $r \in \mathbf{R}$ is a role name and $a_1, \ldots, a_n \in \mathbf{I}$ are individuals. If C_1, C_2 are concepts then $C_1 \sqsubseteq C_2$ is an *inclusion axiom*. If C is a concept, $r \in \mathbf{R}$ is a role, and $a, b \in \mathbf{I}$ are individuals, then $C(a)$ and $r(a, b)$ are *assertional axioms*. An *ontology* $\mathcal{O}$ is a pair $\mathcal{O} = (\mathcal{T}, \mathcal{A})$ where $\mathcal{T}$ is a finite set of inclusion axioms (called the Tbox) and $\mathcal{A}$ is a finite set of assertional axioms (called the Abox). The signature $Sig(\mathcal{O})$ of an ontology $\mathcal{O}$ is the set $Sig(\mathcal{O}) \subseteq Sig_{\mathcal{L}}$ of symbols occurring in $\mathcal{O}$. The signature $Sig(\alpha)$ of an axiom α is defined analogously. If $\mathcal{O} = (\mathcal{T}, \mathcal{A})$ is an ontology and α is an axiom we define $\mathcal{O} \cup \{\alpha\}$ to be either $\mathcal{O} \cup \{\alpha\} = (\mathcal{T} \cup \{\alpha\}, \mathcal{A})$ or $\mathcal{O} \cup \{\alpha\} = (\mathcal{T}, \mathcal{A} \cup \{\alpha\})$, depending on whether α is an inclusion or assertional axiom. The set difference is defined analogously. We assume the standard first-order semantics of $\mathcal{O}$, given by Tarski style model-theoretic semantics using interpretations like in [1].

4 Transactions on Ontologies

In this section, we illustrate the problem of concurrent transaction management for ontologies and specify the notion of atomic operations, transactions, transaction schedules and serializability. By a slight abuse of the notation, we use standard set operators in the context of sequences, e. g., $a \in (a_1, \dots, a_n) \iff a \in \{a_1, \dots, a_n\}$. The union $\cup$ of two sequences is defined as the set $(a_1, \dots, a_n) \cup (b_1, \dots, b_m) = \{a_1, \dots, a_n, b_1, \dots, b_m\}$ and the concatenation $\circ$ of two sequences is defined as the sequence $(a_1, \dots, a_n) \circ (b_1, \dots, b_m) = (a_1, \dots, a_n, b_1, \dots, b_m)$.

Definition 1. *Let $\mathcal{O}$ be an ontology in language $\mathcal{L}$ and $\mathbf{V}_C$, $\mathbf{V}_R$ and $\mathbf{V}_I$ be sets of variable names for concepts, roles and individuals. Then an atomic operation a on $\mathcal{O}$ is a tuple $a = (o, \alpha)$, consisting of one operation $o \in \{ask, tell, forget\}$ and an axiom α with 1.) $\alpha \in \mathcal{L}'$ with $Sig_{\mathcal{L}'} = (\mathbf{C} \cup \mathbf{V}_C) \uplus (\mathbf{R} \cup \mathbf{V}_R) \uplus (\mathbf{I} \cup \mathbf{V}_I)$ (if $o = ask$), 2.) $\alpha \in \mathcal{L}$ (if $o = tell$), or 3.) $\alpha \in \mathcal{O}$ (if $o = forget$).*

For an atomic operation (ask, α), we allow α to contain variable names in order to ask for more general formulas, e. g., the operation $(ask,\ A \sqsubseteq ?X)$ or $(ask,\ ?Y \sqsubseteq B)$ with $?X, ?Y \in \mathbf{V}_C$, asking for axioms with concept descriptions C such that $\mathcal{O} \models A \sqsubseteq C$ is true or for all axioms with concept descriptions D that $\mathcal{O} \models D \sqsubseteq B$ is true, respectively. Hence, an empty result to an *ask* operation means *false* whereas some result would mean *true*. The operation $(forget, \alpha)$ triggers a *contraction* of $\mathcal{O}$ by $\alpha \in \mathcal{O}$ yielding a new ontology $\mathcal{O}' = \mathcal{O} \setminus \{\alpha\}$. The operation $(tell, \alpha)$ triggers an expansion of $\mathcal{O}$ by α yielding a new ontology $\mathcal{O}' = \mathcal{O} \cup \{\alpha\}$. Note that we do not consider the general problem of complex belief dynamics in ontologies [12]. For example, we do not consider the problem of *revising* an ontology by a possibly contradicting axiom α such that the new ontology remains consistent. To formalize the above intuition, we introduce two functions that describe the results of an atomic operation. *ans*—answer, returns a set of axioms for a given pair of ontology and atomic operation—and *upd*—update returns a new (updated) ontology, for a given pair of ontology and atomic operation. For $\alpha \in \mathcal{L}'$ with $Sig_{\mathcal{L}'} = (\mathbf{C} \cup \mathbf{V}_C) \uplus (\mathbf{R} \cup \mathbf{V}_R) \uplus (\mathbf{I} \cup \mathbf{V}_I)$ let $gr(\alpha)$ be the set of *groundings* of α in $\mathcal{L}$, i. e., the set of all axioms that are the same as α but every variable is substituted by some concept description, role description, or individual. Let $\mathcal{O}$ be an ontology and $a = (o, \alpha)$ an atomic operation. Then define

$$ans(\mathcal{O}, (o, \alpha)) = \begin{cases} \{\alpha' \in gr(\alpha) \mid \mathcal{O} \models \alpha'\} & o = ask \\ \emptyset & \text{otherwise} \end{cases}$$

$$upd(\mathcal{O}, (o, \alpha)) = \begin{cases} \mathcal{O} & o = ask \\ \mathcal{O} \cup \{\alpha\} & o = tell \\ \mathcal{O} \setminus \{\alpha\} & o = forget \end{cases}$$

Note that only the *ask* operation may yield a non-empty answer and only *tell* and *forget* operations actually update the ontology. Based on these *atomic operations*, we are able to define *ontology transactions* as follows.

Definition 2. *An* ontology transaction θ *(or* transaction *for short) is a finite sequence $\theta = (a_1, \dots, a_n)$ of atomic operations $a_1, \dots, a_n$.*

A transaction bundles a sequence of atomic operations to be executed on behalf of a user. For a transaction $\theta = ((o_1, \alpha_1), \dots, (o_n, \alpha_n))$ let $\mathsf{axioms}(\theta) = \{\alpha_1, \dots, \alpha_n\}$. We denote with $Sig(\theta) \subseteq Sig_{\mathcal{L}'}$ the signature of all axioms of a transaction θ, with $\mathcal{L}'$ being the same as in Definition 1.

For an ontology $\mathcal{O}$ and a sequence of atomic operations $(a_1, \dots, a_n)$, in order to take cumulative changes of $\mathcal{O}$ into account, we abbreviate

$$upd(\mathcal{O}, ()) = \mathcal{O} \tag{1}$$

$$upd(\mathcal{O}, (a_1, \dots, a_n)) = upd(upd(\mathcal{O}, (a_1, \dots, a_{n-1})), a_n) \tag{2}$$

for all $i = 1, \dots, n$. In other words, $upd(\mathcal{O}, (a_1, \dots, a_n))$ is the ontology resulting after sequentially executing the atomic operations $a_1, \dots, a_n$. Analogously, $ans(\mathcal{O}, a_n)$ is the answer of the atomic operation a_n on $upd(\mathcal{O}, (a_1, \dots, a_{n-1}))$.

$$ans(\mathcal{O}, a_n) = ans(upd(\mathcal{O}, (a_1, \dots, a_{n-1})), a_n) \tag{3}$$

Example 2. To clarify this, we continue with formalizing our Example 2 from the introduction according to the definitions made so far. Let $\theta_1 = (a_1, a_2), \theta_2 = (b_1)$ be the two transactions on $\mathcal{O}$ defined as:

$$a_1 = (forget, B \sqsubseteq D_1) \qquad a_2 = (tell, B \sqsubseteq D_2)$$
$$b_1 = (ask, A \sqsubseteq ?X)$$

As defined in the introduction, transaction θ_1 intends to replace the axiom $B \sqsubseteq D_1$ by $B \sqsubseteq D_2$ while transaction θ_2 asks for all subsumption relations of the form $A \sqsubseteq ?X$. Figure 4 shows the interaction between these two transactions. The left part of the figure shows transaction θ_1, while the right part shows the three possible execution orders of transactions θ_1 and θ_2. As we can see, depending on the transaction order, the outcome differs. The outcome of the operation of $b_1 = (ask, (A \sqsubseteq ?X))$ is $A \sqsubseteq A_1$ if the operation takes place before and $A \sqsubseteq A_2$ after the operations of θ_1, a_1, a_2. Both cases refer to a serial schedule. However, in the second case, we observe unintended answers of transaction θ_2. Since the operation (b_1) takes place in between the operations a_1 and a_2 (non-serial schedule) the only concept that subsumes A is the universal concept $\top$.

Definition 3. *Let $\Theta = \{\theta_1, \dots, \theta_k\}$ with $\theta_i = (a_{i,1}, \dots, a_{i,m_i})$ for $i = 1, \dots, k$ be a set of transactions. A* transaction schedule *π of Θ is a transaction $\pi = (c_1, \dots, c_m)$ such that*

$$\{c_1, \dots, c_m\} = \theta_1 \cup \theta_2 \cup \ldots \cup \theta_k \tag{4}$$

and for all $i = 1, \dots, k$ we have $u < r$ iff $s < t$ for $c_u = a_{i,s}$ and $c_r = a_{i,t}$.

Definition 4. *Let $\Theta = \{\theta_1, \dots, \theta_k\}$ be a set of transactions. A transaction schedule π is a* serial transaction schedule *of Θ if there is a permutation $\sigma : \{1, \dots, k\} \to \{1, \dots, k\}$ such that*

$$\pi = \theta_{\sigma(1)} \circ \theta_{\sigma(2)} \circ \ldots \circ \theta_{\sigma(k)} \tag{5}$$

Let Π_{ser_Θ} be the set of all possible serial transaction schedules *for a given set of transactions Θ.*

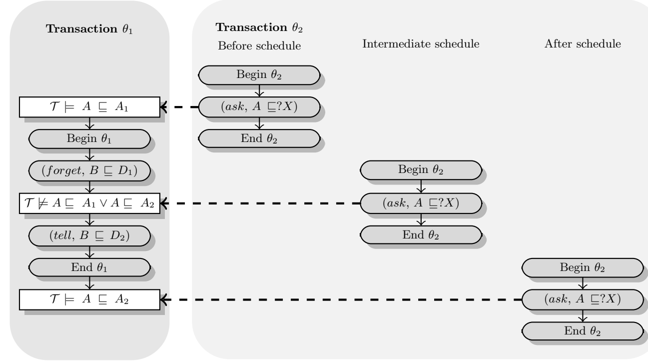

Fig. 1. Transaction Processing

Obviously, a *serial transaction schedule* π_{ser} is a transaction schedule that respects the original order of the atomic operations in the original transactions and executes operations of the individual transactions in distinguishable batches. For a set Θ of n transactions θ_i with $i = 1, \ldots, n$ there exist $n!$ different *serial transaction schedules*. Apart from the *serial transaction schedules*, a vast number of other interleaving schedules exists, e. g., for two transactions of lengths m_1, m_2 the possible number of schedules is $\binom{m_1+m_2}{m_1}$. So there is, in general, a large number of possibilities for transactions to interleave. In order to both preserve the intended semantics of a set of transactions and optimizing performance we are interested in *serializable* schedules.

Definition 5. *Let $\mathcal{O}$ be an ontology and $\Theta = \{\theta_1, \ldots, \theta_k\}$ be a set of transactions on $\mathcal{O}$. A transaction schedule $\pi' = (c_1, \ldots, c_n)$ of Θ is* serializable *if there exists a serial transaction schedule $\pi_{ser} = (d_1, \ldots, d_n)$ such that $c_i = d_{\sigma(i)}$ (for $i = 1, \ldots, n$) for some bijection $\sigma : \{1, \ldots, n\} \to \{1, \ldots, n\}$ of Θ such that*

1. *$upd(\mathcal{O}, \pi') = upd(\mathcal{O}, \pi_{ser})$*
2. *$ans(\mathcal{O}, (c_1, \ldots, c_i)) = ans(\mathcal{O}, (d_{\sigma(1)}, \ldots, d_{\sigma(i)}))$ for $i = 1, \ldots, n$*

In other words, a transaction schedule π' is serializable wrt. $\mathcal{O}$ if there is a *serial transaction schedule* π_{ser} such that applying π' on $\mathcal{O}$ yields the same ontology as applying π_{ser} on $\mathcal{O}$ and all answers to queries stay the same.

Example 3. We continue Example 2 with the two transactions $\theta_1 = (a_1, a_2)$ and $\theta_2 = (b_1)$. Possible transaction schedules, which adhere to a fixed order in operations of the same transaction, are $\pi_1 = (a_1, a_2, b_1)$, $\pi_2 = (a_1, b_1, a_2)$ and $\pi_3 = (b_1, a_1, a_2)$. Both π_1 and π_3 are serial transaction schedules. The schedule π_2 is not serializable due to

$$ans_{\pi_1}(\mathcal{O}, b_1) = \{A \sqsubseteq A_1\} \qquad ans_{\pi_3}(\mathcal{O}, b_1) = \{A \sqsubseteq A_2\}$$
$$ans_{\pi_2}(\mathcal{O}, b_1) \cap \{A_1, A_2\} = \emptyset$$

Definition 6. *A set of transaction $\Theta = \{\theta_1, \ldots, \theta_n\}$ is* conflicting *if there is a transaction schedule π that is not serializable.*

Obviously, in the case of conflicting transactions, some mechanism need to decide how transactions have to be scheduled in order to have a well-defined outcome of concurrent transactions. In the following, we address this issue in a conservative way by restricting interleaving executions of possibly conflicting transactions using locking.

5 What has to be Locked?

A problem of concurrent transaction management is to find, if existing, a *serializable transaction schedule* for a sequence of transactions $\theta_1, \ldots, \theta_n$. The simplest *serializable transaction schedule* for a given set of transactions Θ would be a *serial transaction schedule*, i. e., locking the whole ontology. However, such a schedule would potentially suffer from execution delays regarding multiple transactions. The other extreme is to lock exactly this part of the ontology necessary to avoid conflicts, but this could suffer from a potentially expensive calculation of the concrete locking area. To remedy this trade-off, we investigate the problem of determining the right part of the ontology that has to be locked. Based on this, we are able to investigate the problem of acquiring locks and determining a *serializable interleaving transaction schedule.*

5.1 Modules of an Ontology

According to our example in Sect. 4, a lock has to be acquired on more than just the axioms of the operations of a transaction ($\mathsf{axioms}(\theta)$). Additionally, also the logical consequences, constructed using symbols from $Sig(\theta)$, should be locked. Thus, for a transaction θ over ontology $\mathcal{O}$, we have to lock a sub-ontology $\mathcal{O}_\theta \subseteq \mathcal{O} \cup \mathsf{axioms}(\theta)$, so that every logical consequence α constructed using *only* symbols from $Sig(\theta)$ with $\mathcal{O} \cup \mathsf{axioms}(\theta) \models \alpha$ is already a logical consequence of $\mathcal{O}_\theta$. It is possible to define finite sets of axioms $\mathcal{M} \subseteq \mathcal{O}$ such that for all axioms α with terms only from some Signature $\mathbf{S} \subseteq Sig(\mathcal{O})$, we have that $\mathcal{M} \models \alpha$ iff $\mathcal{O} \models \alpha$. In such case $\mathcal{M}$ is called $\mathbf{S}$-module of $\mathcal{O}$, cf. [6].

Definition 7. *Let $\mathcal{O}' \subseteq \mathcal{O}$ be ontologies and $\mathbf{S}$ be a signature. Then $\mathcal{O}'$ is a module for $\mathbf{S}$ of $\mathcal{O}$, if for all axioms α with $Sig(\alpha) \subseteq \mathbf{S}$, it holds that $\mathcal{O}' \models \alpha$ if and only if $\mathcal{O} \models \alpha$.*

An important property of modules is *convexity*, i. e., given three ontologies $\mathcal{O}_1 \subseteq \mathcal{O}_2 \subseteq \mathcal{O}_3$ if $\mathcal{O}_1$ is an $\mathbf{S}$-module in $\mathcal{O}_3$ then $\mathcal{O}_1$ is an $\mathbf{S}$-module in $\mathcal{O}_2$ and $\mathcal{O}_2$ is an $\mathbf{S}$-module in $\mathcal{O}_3$ [6]. This means that it is sufficient to focus on minimal $\mathbf{S}$-modules. An $\mathbf{S}$-module $\mathcal{O}_1$ is minimal if there is no other $\mathbf{S}$-module $\mathcal{O}_2 \subsetneq \mathcal{O}_1$. This is also advantageous from a locking point of view, locking less is better since is is more likely that other transactions could also be executed. However, just one module is not enough since for a given signature $\mathbf{S}$ and an ontology

$\mathcal{O}$ there might be multiple **S**-modules and for our task we are interested in the fragment $\mathcal{O}_\theta$ that covers all axioms essential for the transaction θ. For such kind of fragment of an ontology the literature gives us the following definition, cf. [6].

Definition 8. *For a signature* $\mathbf{S}$ *and an ontology* $\mathcal{O}$*, we say that an axiom* $\alpha \in \mathcal{O}$ *is* $\mathbf{S}$*-essential in* $\mathcal{O}$ *wrt.* $\mathcal{L}$ *if* α *belongs to some minimal* $\mathbf{S}$*-module in* $\mathcal{O}$ *wrt.* $\mathcal{L}$*.*

Unfortunately, it has been shown in the literature that deciding if a set of axioms is a module is hard or even undecidable for expressive DLs [6, 7]. But there exists several alternative (approximative) definitions of modules. One of them is the so called locality-based module (LBM) [16], which comes in two flavors, syntactic and semantic LBM. For syntactic LBMs it is known that they contain the corresponding semantic LBM and for their calculation algorithms with polynomial runtime wrt. the size of the ontology are known [16].

Based on the definition above, we can now state our notion of *influence area*, which describes the set of all axioms and entailments, which could be influenced by a single *atomic operation*.

Definition 9. *The minimal influence area* Ω_a *of an atomic operation* $a = (o, \alpha)$ *with respect to an ontology* $\mathcal{O}$ *is the set of all* $Sig(\alpha)$*-essential axioms in* $\mathcal{O}$*. If* $o = tell$ *we extend the definition to all* $Sig(\alpha)$*-essential axioms in* $\mathcal{O} \cup \{\alpha\}$*.*

5.2 Two-Phase Locking for Ontologies

Now, we are able to define a 2PL based locking mechanism for ontology transactions. Algorithm 1 displays the locking procedure. The input to the algorithm is

Algorithm 1: ExecuteTransaction

```
   input : θ, a single transaction, GLock a globale syncronized Lock
 1 begin
       /* Initialization                                          */
 2     TLock ⟵ ∅;
       /* Expanding phase: acquire locks                          */
 3     for i = 1 to |θ| do
 4         a ⟵ θ[i];
 5         while ((GLock\TLock) ∩ Ω_Sig(a) ≠ ∅) do
 6             wait;
 7         GLock ⟵ GLock\TLock;
 8         TLock ⟵ Ω_Sig(a_1...a_i);
 9         GLock ⟵ GLock ∪ TLock;
10         execute a;
       /* Shrinking phase: remove all locks                       */
11     GLock ⟵ GLock\TLock;
```

the transaction θ_i and a global lock $GLock$, which is synchronized for all running

instances of this procedure. The algorithm can be subdivided into three parts. First the initialization part, in which a local empty lock $TLock$ is initialized, line (2). The second part of the algorithm complies the 'Expanding Phase' of the *2PL* mechanism. The algorithm picks the current atomic operation (a) (4). Only if the intersection between this Ω_a and the global lock $GLock$ is empty the algorithm will continue, otherwise it will wait (5,6). During the time procedure (a) is waiting for resources to be freed, it could happen that another parallel working procedure (b) changes the ontology in two ways that could affect (a). First, an axiom currently in the $TLock$ of (a) is removed by (b), then the $TLock$ of (a) is just too big but the locking is still valid. Second, a new axiom that should be part of $TLock$ (a) is added by (b), then the calculated $TLock$ of (a) is to small and therefore it has to be constantly recalculated. If it is empty the procedure acquires the lock for a, by adding Ω_a to $TLock$ as well as to $GLock$, lines (7,8,9). Then the procedure could execute the *atomic operation* a, line (10). As soon as all *atomic operations* of θ_i are processed, the procedure enters the 'Shrinking Phase' (third part) and frees all acquired locks (line (11)).

Theorem 1. *Let $\Theta = \{\theta_1, \ldots, \theta_n\}$ be a set of transactions. Any transaction schedule that is emitted by parallel executions of Algorithm 1 for each transaction $\theta_1, \ldots, \theta_n$ is serializable.*

Proof (Sketch). Let $\Theta = \{\theta_1, \ldots, \theta_n\}$ with $\theta_i = (a_{i,1}, \ldots, a_{i,m_i})$ for $i = 1, \ldots, k$ be a set of transactions. Let $\pi = (c_1, \ldots, c_n)$ be a transaction schedule emitted by the parallel executions of Algorithm 1. Consider the serial transaction schedule $\pi_{ser} = \theta_{\sigma(1)} \circ \ldots \theta_{\sigma(n)}$ with a permutation $\sigma : \{1, \ldots, n\} \to \{1, \ldots, n\}$ and $\sigma(i) < \sigma(j)$ iff $k < l$ for $c_k = a_{i,1}$ and $c_l = a_{j,1}$. In other words, π_{ser} is the serial schedule obtained from π by ordering the transactions according to their first operation in π. It suffices to show that π_{ser} is the witness of π's serializability according to Definition 5. Assume $upd(\mathcal{O}, \pi') = upd(\mathcal{O}, \pi_{ser})$ does not hold. Then there are transactions θ, θ' that manipulate some axiom $\alpha \in \mathcal{O}$. Without loss of generality assume θ appears before θ' in π_{ser}. Then θ acquires a lock on at least the axiom α—note that always $\alpha \in \Omega_{Sig(\alpha)}$—in line 9 of Algorithm 1 and releases it only after executing the whole transaction in line 11. Then θ' is blocked and $\mathcal{O}$ is updated in the same way as a serial execution of θ and θ', as in π_{ser}. It follows $upd(\mathcal{O}, \pi') = upd(\mathcal{O}, \pi_{ser})$. Similarly, it also holds that the answer behavior is the same for both π and π_{ser} by taking into account that the subset $\Omega'_{Sig(\alpha)} \subseteq \mathcal{O}$ that suffices to produce answers for an operation (ask, α)—i. e. $ans(\mathcal{O}, (ask, \alpha)) = \{\alpha' \in gr(\alpha) \mid \mathcal{O} \models \alpha'\} = \{\alpha' \in gr(\alpha) \mid \Omega_{Sig(\alpha)} \models \alpha'\}$— is accessed by only one transaction at a time as well. □

6 Evaluation

For our evaluation, we use different versions of the National Cancer Institute Thesaurus (NCIt) which are available as OWL EL++ ontologies[1]. As there are

[1] NCIt archive `http://evs.nci.nih.gov/ftp1/NCI_Thesaurus/archive`, Nov 2012

no real transaction logs available for NCIt (or any other versioned ontology), we perform our evaluation using transactions artificially generated from four consecutive versions available for NCIt. More specifically, for each two consecutive versions of the NCIt ontology, we generate around 140 different transactions, each consisting of 6-12 atomic operations, which contain *tell*-operations on axioms that are present in the more recent version but missing in the previous version, *forget*-operations on axioms that are present in the previous version but missing in the more recent version, and *ask*-operations on axioms artificially generated partially from the signature of the *tell*- and *forget*-operations in the same transaction and potentially other symbols. We computed schedules for around 240 different combinations of these transaction.

Our evaluation aims at measuring the potential benefit of the module-based locking approach in terms of total execution time. For each atomic operation in a transaction, we compute the locking areas based on syntactic locality as described in Sec. 5. While the time needed for computing a module-based lock is, in general, much larger than for the whole ontology (which is almost immediate) we estimate a benefit when taking varying execution times of *non-critical* operations—i. e. user code that is contained in a transaction—into account. We expect that with increasing average execution time of non-critical operations the effort for computing a more specific locking area becomes negligible.

6.1 Evaluation Setup

In order to compensate for the lack of existing real transaction logs, we implemented Algorithm 1 in a non-parallel fashion and compute all serializable transaction schedules that are consistent with our locking approach. Let Θ^{mod} resp. Θ^{onto} be these sets of serializable transaction schedules. For the approach of locking the whole ontology for each atomic operation it follows that Θ^{onto} is the set of all serial transaction schedules. For both locking approaches and each transaction schedule $\theta = (c_1, \ldots, c_n)$ obtained in this way, we estimate the running time for executing the schedule as follows. Each atomic operation c_i $(i = 1, \ldots, n)$ can be decomposed via $c_i = c'_i c''_i c'''_i$, where in c'_i the lock is acquired—which might take some time of lock calculation and the locking itself—c''_i is the *critical operation*—which contains the actual database access and is the reason for acquiring the lock—and c'''_i is a *non-critical operation*, which might contain user interaction and other user code. For each non-critical operation c'''_i, we consider different (but uniform over all non-critical operations) execution times while we assume critical operations to be immediate, i. e. they have an execution time of zero. If θ contains a sequence $c_i c_{i+1} = c'_i c''_i c'''_i c'_{i+1} c''_{i+1} c'''_{i+1}$ where c_i and c_{i+1} originate from different transactions we assume that c'''_i and $c'_{i+1} c''_{i+1} c'''_{i+1}$ can be executed in parallel, thus decreasing total execution time. A *parallelization* f_θ of θ is a function $f_\theta : \{1, \ldots, n\} \to \{1, \ldots, n\}$ that satisfies

1. $f_\theta(i) \leq f_\theta(j)$ for all $i, j = 1, \ldots, n$,
2. If $f_\theta(i) = f_\theta(j)$ then c''_i and c''_j come from different transactions, and
3. there is $n' \leq n$ with $\mathsf{Im}\ f_\theta = \{1, \ldots, n'\}$ ($\mathsf{Im}\ f$ is the image of a function f)

Therefore, a parallelization f_θ says that all c_i'' with $f_\theta(i) = 0$ are executed in parallel at a first step (after their corresponding c_i'). Then all c_i'' with $f_\theta(i) = 1$ are executed in parallel, and so on. The first requirement above ensures that no c_i is executed before c_j if $j < i$. The second requirement says that only operations of different transactions can be executed in parallel, and the third requirement states that there are no steps in the execution where nothing is executed. Due to the assumed execution time of zero for critical operations, we can neglect those. Let F_θ be the set of all parallelizations of θ. As there may be different variants on how to parallelize a single transaction schedule we average the total execution time over all of them. Let t_{nc} be the average execution time for a non-critical operation and let $t_X(\theta)$ be the total time needed for computing locks in θ wrt. the approach $X \in \{\text{onto}, \text{mod}\}$. Then we estimate the total execution time for a transaction schedule $\theta = (c_1, \ldots, c_n)$ via

$$T^X_{t_{nc}}(\theta) = t_X(\theta) + t_{nc} \frac{\sum_{f_\theta \in F_\theta} \max \mathsf{Im}\ f_\theta}{|F_\theta|}$$

Finally, for each t_{nc} we take the average total execution time over all transaction schedules for both approaches, i. e.

$$T^X_{t_{nc}} = \frac{\sum_{\theta \in \Theta^X} T^X_{t_{nc}}(\theta)}{|\Theta^X|}$$

with $X \in \{\text{onto}, \text{mod}\}$. The implementation used for our evaluation can be downloaded from `https://launchpad.net/ontotrans`.

6.2 Results

As mentioned, we considered different combinations of transactions of different lengths. For around 30% of these tested combinations (≈ 240 combinations), we could find serializable interleaving schedules. This seams to be strongly related to our strategy of randomly picking axioms to generate the operations of a transaction. The influence area of a whole transaction consisting of randomly generated operations can be quite large so that the only possible serializable schedules for a combination of such transactions are the serial ones. For real transactions, we assume the axioms in the single operations to be more related to each other and therefore the influence areas to be smaller. Due to reasons of execution time, we decided to compute a maximum of 30 schedules per tested transaction combination. With these settings, we were able to find around 1200 serializable transaction schedules. The average serializable transaction schedules has only 76.642% of the length of the serial schedules and a single computation of the two modules, one for the global lock and one for the current atomic operation takes in average 2.832 seconds. Figure 2, displays the average total execution time for a schedule of average length of ten, considering different execution times for the non-critical part c_i''' of the atomic operation. The figure shows that starting from a average execution time for a non-critical operations of around 12 seconds the locking based approach starts to perform better.

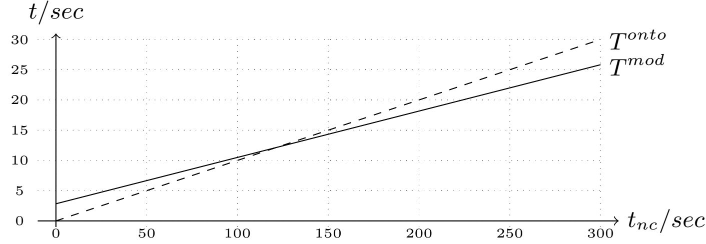

Fig. 2. Average Total Execution Time for T^{mod} vs. T^{onto}

6.3 Lessons Learned and Discussion

The relatively high threshold shown in Fig. 2 is the result of the expensive module calculation. Due to a lack of implementations of incremental module calculation mechanisms like those introduced in [17], we use the locality-based module calculation of the OWLAPI which recalculates the global module for every comparison. It turns out that this global lock calculation takes on average over 90% of the whole time spend on module calculation. Thus, applying an optimized incremental module calculation and efficient caching strategies would lead to a significant decrease in average module calculation time and therefore to a significantly lower threshold. However, even with our naive implementation our results depicted in Fig. 2 clearly show the benefit of computing module-based locks as total execution time decreases compared to the naive approach.

7 Conclusion

In this paper, we have presented a locking approach for concurrent ontology transactions. While the management of transactions in general is a challenging problem on its own, it becomes more complicated for ontologies since changes in an ontology also affect the entailments of the ontology. Thus, the management of transactions has to take the entailments of an ontology into account. Several research has been done in order to analyze changes in ontologies and to compare versions of ontologies or to build links between ontology versions. The locking approach in this paper is a further step towards collaborative ontology management. The locking principle takes the dependencies between axioms regarding the DL entailment into account, by determining the influence area of transactions. Locking policies lock ontologies according to the influence area of a transaction.

As a next step, we plan to investigate efficient scheduling of ontology transactions, while the presented locking principles and locking policies are the fundamental building blocks of a scheduling approach.

Acknowledgments The research reported here was partially supported by the SocialSensor FP7 project (EC under contract number 287975).

References

1. F. Baader, D. Calvanese, D. L. McGuinness, D. Nardi, and P. F. Patel-Schneider, editors. *The Description Logic Handbook*. Cambridge University Press, 2003.
2. P.A. Bernstein and E. Newcomer. *Principles of Transaction Processing*. Morgan Kaufmann, 1997.
3. Philip A. Bernstein, Vassos Hadzilacos, and Nathan Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987.
4. Vinay K. Chaudhri, Vassos Hadzilacos, and John Mylopoulos. Concurrency Control for Knowledge Bases. In Bernhard Nebel, Charles Rich, and William R. Swartout, editors, *KR*, pages 762–773. Morgan Kaufmann, 1992.
5. S. M. Falconer, T. Tudorache, and N. F. Noy. An analysis of collaborative patterns in large-scale ontology development projects. In *K-CAP*, pages 25–32. ACM, 2011.
6. B. C. Grau, I. Horrocks, Y. Kazakov, and U. Sattler. Modular Reuse of Ontologies: Theory and Practice. *Journal of Artificial Intelligence Research*, 31:273–318, 2008.
7. Bernardo Cuenca Grau, Ian Horrocks, Yevgeny Kazakov, and Ulrike Sattler. Extracting Modules from Ontologies: A Logic-Based Approach. In Heiner Stuckenschmidt, Christine Parent, and Stefano Spaccapietra, editors, *Modular Ontologies*, volume 5445 of *LNCS*, pages 159–186. Springer, 2009.
8. P. Hitzler, M. Krötzsch, B. Parsia, P. F. Patel-Schneider, and S. Rudolph. OWL 2 Web Ontology Language Primer. *W3C Recommendation*, 27:1–123, 2009.
9. E. Jiménez-Ruiz, B. C. Grau, I. Horrocks, and R. B. Llavori. Supporting concurrent ontology development: Framework, algorithms and tool. *Data Knowl. Eng.*, 70(1):146–164, 2011.
10. P. D. Karp, V. K. Chaudhri, and S. M. Paley. A Collaborative Environment for Authoring Large Knowledge Bases. *J. Intell. Inf. Syst.*, 13(3):155–194, 1999.
11. Natalya Fridman Noy and Mark A. Musen. Specifying Ontology Views by Traversal. In Sheila A. McIlraith, Dimitris Plexousakis, and Frank van Harmelen, editors, *ISWC*, volume 3298 of *LNCS*, pages 713–725. Springer, 2004.
12. Guilin Qi and Fangkai Yang. A Survey of Revision Approaches in Description Logics. In Diego Calvanese and Georg Lausen, editors, *RR*, volume 5341 of *LNCS*, pages 74–88. Springer, 2008.
13. Julian Seidenberg and Alan Rector. A methodology for asynchronous multi-user editing of semantic web ontologies. In Derek H. Sleeman and Ken Barker, editors, *K-CAP*, pages 127–134. ACM, 2007.
14. Julian Seidenberg and Alan L. Rector. Web ontology segmentation: analysis, classification and use. In Les Carr, David De Roure, Arun Iyengar, Carole A. Goble, and Michael Dahlin, editors, *WWW*, pages 13–22. ACM, 2006.
15. Tania Tudorache, Sean M. Falconer, Natalya F. Noy, Csongor Nyulas, Tevfik Bedirhan stn, Margaret-Anne D. Storey, and Mark A. Musen. Ontology development for the masses: Creating icd-11 in webprotg. In P. Cimiano and H. S. Pinto, editors, *EKAW*, volume 6317 of *LNCS*, pages 74–89. Springer, 2010.
16. C. Del Vescovo, P. Klinov, B. Parsia, U. Sattler, T. Schneider, and D. Tsarkov. Syntactic vs. semantic locality: How good is a cheap approximation? *CoRR*, abs/1207.1641, 2012.
17. C. Del Vescovo, B. Parsia, U. Sattler, and T. Schneider. The modular structure of an ontology: Atomic decomposition and module count. In *WoMO*, volume 230, pages 25–39. IOS Press, 2011.