Ontologies have been applied in a wide range of practical domains. They play a key role in data modeling, information integration, and the creation of semantic web services, intelligent web sites and intelligent software agents. The Web ontology language OWL, recommended by W3C, is based on description logics (DLs). Automated reasoning in DLs is very important for the success of OWL, as it provides support for visualization, debugging, and querying of ontologies. The existing ontology reasoners are not yet satisfactory, especially when dealing with qualified number restrictions and large ontologies. In this paper, we present the design of our new reasoner TGC2, which uses tableaux with global caching for reasoning in $\mathcal{E}x\mathcal{T}\mathcal{U}\mathcal{N}_\mu$-complete DLs. The characteristic of TGC2 is that it is based on our tableau methods with the optimal ($\mathcal{E}x\mathcal{T}\mathcal{U}\mathcal{N}_\mu$) complexity, while the existing well-known tableau-based reasoners for DLs have a non-optimal complexity (at least $\mathcal{N}\mathcal{E}x\mathcal{T}\mathcal{U}\mathcal{N}_\mu$). We briefly describe the tableau methods used by TGC2. We then provide the design principles of TGC2 and some important optimization techniques for increasing the efficiency of this reasoner. We also present preliminary evaluation results of TGC2. They show that TGC2 deals with qualified number restrictions much better than the other existing reasoners.

Keywords: Description logics; automated reasoning; optimization techniques.

1. Introduction

An ontology is a formal description of a common vocabulary for a domain of interest and the relationships between terms built from the vocabulary. Ontologies have been applied in a wide range of practical domains such as medical informatics, bio-informatics and the semantic web. They play a key role in data modeling, information integration, and the creation of semantic web services, intelligent web sites
and intelligent software agents. Ontology languages based on description logics (DLs) like OWL 2 are becoming increasingly popular due to the availability of DL reasoners, which provide automated support for visualization, debugging, and querying of ontologies.

Ontology classification and consistency checking of ontologies are basic reasoning services of ontology reasoners. The goal of classification is to compute the hierarchical relation between concepts, which is used, e.g., for browsing ontologies, for user interface navigation, and for suggesting more (respectively, less) restricted queries. Consistency checking is a very important task for ontology engineering. It is essentially the problem of checking satisfiability of the knowledge base (KB) representing the ontology in DL. Most of other reasoning problems for ontologies, including instance checking and ontology classification, are reducible to this one.

The most efficient decision procedures for the mentioned problems are usually based on tableaux [1], as many optimization techniques have been developed for them. However, the satisfiability problem in DLs has high complexity (ExpTime-complete in $\mathcal{ALC}$ and $\mathcal{N2ExpTime}$-complete in $\mathcal{SRIOIQ}$; see, e.g., http://www.cs.man.ac.uk/~ezolin/dl/) and the existing ontology reasoners (including tableau-based ones) are not yet satisfactory, especially when dealing with qualified number restrictions and large ontologies.

According to the survey [1], the third generation ontology reasoners that support $\mathcal{SHIQ}$ or $\mathcal{SRIOIQ}$ are RACER, HermiT, FaCT++, Pellet and KAON2. The reasoners RACER, Pellet and FaCT++ are based on tableaux, HermiT is based on hypertableaux, and KAON2 is based on a translation into disjunctive datalog. That is, all of the listed reasoners except KAON2 are tableau-based. According to [2], KAON2 provides good performance for ontologies with rather simple TBoxes, but large ABoxes; however, for ontologies with large and complex TBoxes, the existing tableau-based reasoners still provide superior performance. The current version of KAON2 does not support nominals and cannot handle large numbers in cardinality statements.

The reasoners RACER, FaCT++ and Pellet use traditional tableau decision procedures like the ones for $\mathcal{SHIQ}$ [3] and $\mathcal{SRIOIQ}$ [4]. These procedures use backtracking to deal with disjunction (e.g., the union constructor). Their search space is an “or”-tree of “and”-structures. Despite advanced blocking techniques (e.g., anywhere blocking), their complexities are non-optimal (e.g., $\mathcal{NExpTime}$ or $\mathcal{N2ExpTime}$ instead of $\mathcal{ExpTime}$ for $\mathcal{ALC}$, $\mathcal{SHI}$, $\mathcal{SHIQ}$, $\mathcal{SHIO}$ and $\mathcal{SHOIQ}$). The decision procedure of HermiT also has a non-optimal complexity [5].

The tableau decision procedure given in [6] for $\mathcal{ALC}$ uses a combination of blocking and caching techniques and is complexity-optimal ($\mathcal{ExpTime}$). A more general technique for developing complexity-optimal tableau decision procedures is based on global caching. Global caching means that, instead of a search space of the form “or”-tree of “and”-structures, we use a single “and–or” graph whose nodes
are labeled by unique sets of formulas. The idea of global caching comes from Pratt’s paper on propositional dynamic logic (PDL) [7].

In this work, we present the design of our new reasoner TGC2, which uses tableaux with global caching for reasoning in a large class of modal and DLs, including \( \text{ALC}, \text{SH}, \text{SHI}, \text{SHIQ}, \text{SHIO}, \text{SHOQ}, \) regular grammar logics (REG), \( \text{REG} \) with converse \( (\text{REG}^c) \), \( \text{PDL} \), \( \text{PDL} \) with converse \( (\text{CPDL}) \), and the combination \( \text{CPDL} \) and \( \text{REG}^c \) \( (\text{CPDL}_{\text{reg}}) \). It is designed to work also for some combinations of the listed logics, including the combination of \( \text{SHIQ}, \text{SHIO}, \text{SHOQ} \) and \( \text{REG} \) in the case when there are no interactions between \( O \) (nominals), \( I \) (inverse roles) and \( Q \) (qualified number restrictions). The reasoner does not fully support \( \text{SHOIQ} \) and \( \text{SHROIQ} \). The core reasoning problem supported by TGC2 is to check whether a KB is satisfiable. Other reasoning problems like instance checking or computing the hierarchical relation between concepts are reducible to that satisfiability problem. TGC2 is based on \( \text{ExpTime} \) tableau methods using global caching and its complexity is \( \text{ExpTime} \) when numbers (in qualified number restrictions) are coded in unary.

This work is a revised and extended version of the conference paper [8]. In comparison with the state reported in [8], we have done the implementation to obtain the first version of TGC2 [9] and conducted a preliminary evaluation for TGC2. The experimental results are included in Sec. 7. Additionally, the current paper also contains a formal specification of the DLs and reasoning problems supported by TGC2, as well as a concrete plan of extending and improving this reasoner, with a discussion on further suitable optimizations.

The rest of this paper is structured as follows. In Sec. 2, we introduce notation and semantics of DLs as well as basic reasoning problems in DLs. In Sec. 3, we briefly describe the tableau methods used by TGC2. In Secs. 4 and 5, we present the design principles of TGC2 and some important optimization techniques for increasing the efficiency of this reasoner. In Sec. 6, we inform the current state of the implementation of TGC2 and outline a plan of extending and improving it. Section 7 contains preliminary evaluation results of TGC2 in comparison with the other existing reasoners. Section 8 concludes this work.

2. Preliminaries

The aim of this section is to specify the DLs and reasoning problems supported by TGC2. Our language uses a countable set \( \mathbf{C} \) of concept names, a countable set \( \mathbf{R}_+ \) of role names including a subset of simple role names, and a countable set \( \mathbf{I} \) of individual names. We use letters like \( a, b \) to denote individual names, letters like \( A, B \) to denote concept names, and letters like \( r, s \) to denote role names.

For \( r \in \mathbf{R}_+ \), we call the expression \( r \) the inverse of \( r \). Let \( \mathbf{R}_- = \{ r \mid r \in \mathbf{R}_+ \} \) and \( \mathbf{R} = \mathbf{R}_+ \cup \mathbf{R}_- \). For \( R = \tau \), let \( \overline{R} \) stand for \( r \). We call elements of \( \mathbf{R} \) basic roles and use letters like \( R, S \) to denote them. We define a simple role to be either a simple role name or the inverse of a simple role name.
A context-free semi-Thue system \( \mathcal{S} \) over \( \mathbb{R} \) is a finite set of context-free production rules over alphabet \( \mathbb{R} \). It is symmetric if, for every rule \( R \rightarrow S_1 \ldots S_k \) of \( \mathcal{S} \), the rule \( \mathbb{R} \rightarrow S_1 \ldots S_k \) is also in \( \mathcal{S} \). It is regular if, for every \( R \in \mathbb{R} \), the set of words derivable from \( R \) using the system is a regular language over \( \mathbb{R} \).

A context-free semi-Thue system is like a context-free grammar, but it has no designated start symbol and there is no distinction between terminal and non-terminal symbols. We assume that, for \( R \in \mathbb{R} \), the word \( R \) is derivable from \( R \) using such a system.

A role inclusion axiom (RIA for short) is an expression of the form \( S_1 \circ \ldots \circ S_k \subseteq R \), where \( k \geq 0 \). In the case \( k = 0 \), the left-hand side of the inclusion axiom stands for the identity binary relation \( \varepsilon \).

A regular RBox is a finite set \( \mathcal{R} \) of RIAs such that

\[
\{ R \rightarrow S_1 \ldots S_k \mid (S_1 \circ \ldots \circ S_k \subseteq R) \in \mathcal{R} \}
\]

is a symmetric regular semi-Thue system \( \mathcal{S} \) over \( \mathbb{R} \) such that if \( R \in \mathbb{R} \) is a simple role, then only words with length 1 or 0 are derivable from \( R \) using \( \mathcal{S} \). We assume that \( \mathcal{R} \) is given together with a mapping \( \mathbf{A} \) that associates every \( R \in \mathbb{R} \) with a finite automaton \( \mathbf{A}_R \) recognizing the words derivable from \( R \) using \( \mathcal{S} \). We call \( \mathbf{A} \) the RIA-automaton-specification of \( \mathcal{R} \).

Let \( \mathcal{R} \) be a regular RBox and \( \mathbf{A} \) the RIA-automaton-specification of \( \mathcal{R} \). For \( R, S \in \mathbb{R} \), we say that \( R \) is a subrole of \( S \) w.r.t. \( \mathcal{R} \), denoted by \( R \sqsubseteq \mathcal{R} S \), if the word \( R \) is accepted by \( \mathbf{A}_S \).

Concepts and roles are defined, respectively, by the following grammar rules, where \( C \in \mathbf{C} \), \( a \in \mathbf{I} \), \( r \in \mathbf{R}_+ \), \( n \) is a non-negative integer and \( S \) is a simple role:

\[
C ::= \top \mid \bot \mid A \mid \{ a \} \mid \neg C \mid C \cap C \mid C \cup C \mid \forall R.C \mid \exists R.C \mid \geq n S.C \mid \leq n S.C
\]

\[
R ::= \varepsilon \mid r \mid [R] \mid R \circ R \mid R \cup R \mid R^* \mid C^?
\]

We use letters like \( C \), \( D \) to denote concepts, and letters like \( R \), \( S \) to denote roles.

A TBox is a finite set of axioms of the form \( C \subseteq D \) or \( C \dashv \vdash D \). An ABox is a finite set of assertions of the form \( a : C \), \( S(a, b) \), \( \neg S(a, b) \), \( a \doteq b \) or \( a \neq b \), where \( S \) is a simple role. We also write \( C(a) \) to denote the assertion \( a : C \). A KB is a tuple \( \langle \mathcal{R}, \mathcal{T}, \mathbf{A} \rangle \), where \( \mathcal{R} \) is an RBox, \( \mathcal{T} \) is a TBox and \( \mathbf{A} \) is an ABox.

An interpretation is a pair \( \mathcal{I} = (\Delta^\mathcal{I}, \cdot^\mathcal{I}) \), where \( \Delta^\mathcal{I} \) is a non-empty set called the domain of \( \mathcal{I} \) and \( \cdot^\mathcal{I} \) is a mapping called the interpretation function of \( \mathcal{I} \) that associates each individual name \( a \in \mathbf{I} \) with an element \( a^\mathcal{I} \in \Delta^\mathcal{I} \), each concept name \( A \in \mathbf{C} \) with a set \( A^\mathcal{I} \subseteq \Delta^\mathcal{I} \), and each role name \( r \in \mathbf{R}_+ \) with a binary relation \( r^\mathcal{I} \subseteq \Delta^\mathcal{I} \times \Delta^\mathcal{I} \). The interpretation function \( \cdot^\mathcal{I} \) is extended to complex concepts and complex roles as shown in Fig. 1, where \( |Z| \) denotes the cardinality of a set \( Z \) and, for two binary relations \( R \) and \( S \), \( R \circ S \) denotes their relational composition.

\(^a\text{In the case } k = 0 \text{, the right-hand side of the production rules stand for the empty word, also denoted by } \varepsilon.\)
Design of the Tableau Reasoner TGC2 for DLs

Given an interpretation $I$ and an axiom/assertion $\varphi$, the satisfaction relation $I \models \varphi$ is defined as shown in Fig. 2. If $I \models \varphi$ then, we say that $I$ validates $\varphi$.

![Fig. 2. The satisfaction relation.](image)

An interpretation $I$ is a model of an RBox $R$, a TBox $T$ or an ABox $A$ if it validates all the axioms/assertions of that "box". It is a model of a KB $\langle R, T, A \rangle$, denoted by $I \models \text{KB}$, if it is a model of $R$, $T$ and $A$.

A KB is satisfiable if it has a model. For a KB, we write KB $\models \varphi$ to mean that every model of KB validates $\varphi$. If KB $\models C(a)$, then we say that $a$ is an instance of $C$ w.r.t. KB. If KB $\not\models (C \sqsubseteq D)$ then we say that $C$ is a subconcept of $D$ w.r.t. KB.

Given a KB $\langle R, T, A \rangle$, the problem of checking KB $\models C(a)$ can be reduced to checking whether the KB $\langle R, T, A \cup \{a : \neg C\} \rangle$ is unsatisfiable. Analogously, the problem of checking KB $\models (C \sqsubseteq D)$ can be reduced to checking whether the KB $\langle R, T, A \cup \{\tau : C \cap \neg D\} \rangle$ is unsatisfiable, where $\tau$ is a fresh (new) individual name. That is, the instance checking problem and the problem of computing the hierarchical relation between concepts are reducible to the problem of checking satisfiability of a KB.

The specified DL is more general than $\text{ALC}$, $\text{SHI}$, $\text{SHIQ}$, $\text{SHIO}$, $\text{SHOQ}$, REG, REG$^c$, PDL, CPDL, and CPDL$_{\text{reg}}$. The reasoner TGC2 accepts inputs that
are KBs and queries in the specified DL. However, it can solve a problem only when the following conditions hold:

- There are no interactions between $O$ (nominals like $\{a\}$), $I$ (inverse roles like $R^{-1}$) and $Q$ (qualified number restrictions like $\geq n \cdot S.C.$, $\leq n \cdot S.C.$). Such interactions may be detected during runtime. Syntactically, we can exclude one of the features $O$, $I$, $Q$.
- When the PDL-like $^*$ role constructor is present, the feature $O$ is absent, and either the feature $Q$ or the feature $I$ is absent.

3. Tableaux with Global Caching

This section concerns the problem of checking satisfiability of a KB $\langle R, T, A \rangle$ by using a tableau method. A tableau with global caching for the basic DL $\mathcal{ALC}$ is an “and–or” graph, where each of its nodes is labeled by a unique set of formulas, which are ABox assertions in the case the node is a “complex” node and concepts in the case when the node is a “simple” node. The formulas in the label of a node are treated as requirements to be realized (satisfied) for the node. They are realized by expanding the graph using tableau rules. At the beginning, the graph contains only the root, which is a complex node with label $A \cup \{a : C \mid C \in T \text{ and } a \text{ is an individual name occurring in } A\}$, assuming that $T$ has been translated to concepts that are treated as “global assumptions”. Expanding a node using a static rule causes the node to become an “or”-node, expanding a node using the transitional rule causes the node to become an “and”-node. Usually, the transitional rule has a lower priority than the static rules. An “and”-node is also called a state, while an “or”-node is called a non-state. States in tableaux with global state caching for advanced DLs like $\mathcal{SHIQ}$ [10] or $\mathcal{SHOIQ}$ [11] are more sophisticated than “and”-nodes due to integer linear constraints (ILCS) used for dealing with qualified number restrictions. For an introduction to the tableau method using global caching and its relationship to the other tableau methods, we refer the reader to [12].

The reasoner TGC2 uses a tableau method that unifies our tableau decision procedures developed recently for CPDL$_{reg}$ [13] and the DLs $\mathcal{SHIQ}$ [10], $\mathcal{SHOIQ}$ [11] and $\mathcal{SHIO}$ [14]. We briefly describe below these tableau decision procedures.

- The tableau decision procedure for CPDL$_{reg}$ given in [13] improves the one of [15] by eliminating cuts to give the first cut-free ExpTime (optimal) tableau decision procedure for CPDL$_{reg}$. This procedure uses global state caching, which modifies global caching for dealing with converse without using cuts. It also uses local caching for non-states of tableaux. As cut rules are “or”-rules for guessing the “future” and they are usually used in a systematic way and generate a lot of branches, eliminating cuts is very important for efficient automated reasoning in CPDL$_{reg}$.
- The tableau decision procedure given in [10] is for checking satisfiability of a KB in the DL $\mathcal{SHIQ}$. It has complexity ExpTime when numbers are coded in unary.
It is based on global state caching and integer linear feasibility checking. Both of them are essential for the procedure in order to have the optimal complexity. Global state caching can be replaced by global caching plus cuts, which still guarantee the optimal complexity. However, we chose global state caching to avoid inefficient cuts although it makes our procedure more complicated. In contrast to Farsiniamarj’s method of exploiting integer programming for tableaux [16], to avoid nondeterminism we only check feasibility of the considered set of ILCS, without finding and using their solutions. As far as we know, we are the first one who applied integer linear feasibility checking to tableaux.

- The tableau decision procedure for the DL $SHOQ$ given in the joint works with Golinska-Pilarek [11] has complexity ExpTime when numbers are coded in unary. The DL $SHOQ$ differs from $SHIQ$ in that nominals are allowed instead of inverse roles. The procedure exploits the method of integer linear feasibility checking [10] for dealing with number restrictions. Without inverse roles, it uses global caching instead of global state caching to allow more cache hits. It also uses special techniques for dealing with nominals and their interaction with number restrictions.

- The work [14] presents the first tableau decision procedure with an ExpTime (optimal) complexity for checking satisfiability of a KB in the DL $SHIO$. It exploits global state caching and does not use blind (analytic) cuts. It uses a new technique to deal with the interaction between inverse roles and nominals.

Recall that $SHIQ$, $SHOQ$, $SHIO$ are the three most well-known expressive DLs with ExpTime complexity. Due to the interaction between the features $I$, $Q$ and $O$, the complexity of the DL $SHOIQ$ is $NE$-ExpTime-complete. The reasoner TGC2 allows KBs that use all the features $I$, $O$, $Q$ together, but it does not fully support the DL $SHOIQ$. In the case of interaction between $I$, $O$ and $Q$, the reasoner tries to avoid the difficulty by exploring other branches in the constructed graph in hope for overcoming the problem.

4. The Design Principles of TGC2

In this section, we present the main principles used for the design of TGC2. They are very important for increasing efficiency and scalability of the reasoner.

4.1. Avoiding costly recomputations by caching

Recall that a tableau with global caching is like an “and–or” graph, where each node is labeled by a unique set of formulas. Furthermore, a node may be re-expanded at most once (when dealing with inverse roles or nominals). By constructing such a tableau in a deterministic way instead of searching the space of the form “or”-tree whose nodes are “and”-structures as in the traditional tableau method, the complexity is guaranteed to be ExpTime. The essence of this is to divide the main task to
appropriate subtasks (like expansion of a node) so that each subtask is executed only once and the result is cached to avoid recomputations. (In the traditional tableau method, nodes in “and”-structures are not cached across branches of the main “or”-tree, which means that some kind of recomputation occurs and that causes the non-optimal complexity.)

We apply the idea of avoiding costly recomputations by caching also to other entities/tasks as described below.

- Formulas are normalized and globally cached. Furthermore, certain tasks involved with formulas like computing the negation, the size or the weight of a formula are also cached in order to avoid recomputations.
- As the reasoner TGC2 allows PDL-like role constructors and roles are similar to formulas, they are also normalized and cached. Tasks like computing the inverse of a role, the set of subroles of a role, the size or the weight of a role are cached to avoid recomputations.
- Due to hierarchies of roles and merging nodes in the constructed graph, an edge outgoing from a state may be labeled by a set of roles instead of a single role. For this reason, sets of roles are also cached. Consequently, tasks like computing the union of two sets of roles or the inverse of an edge label are also cached to avoid recomputations.
- When the considered KB uses qualified number restrictions, a state in the constructed graph may have a set of ILCS. Checking feasibility of such a set may be very costly. Furthermore, different states may have the same ILCS. So, we also cache ILCS’s and the results of their feasibility checking to avoid costly recomputations.

4.2. Memory management

As TGC2 may use an exponential number of nodes, formulas and other entities, efficient memory management is very important. It is a critical matter of the approach. The earlier version TGC [17] for reasoning in the basic DL $\mathcal{ALC}$ manages memory usage very well. It shows that, for hard instances of reasoning in the basic DL $\mathcal{ALC}$, execution time but not memory lack is the cause of not being able to solve the instance. For TGC2, in hope for satisfying this claim, we also pay a special effort in memory management.

Caching is good for saving execution time, but it may increase memory usage. On one hand, when identical objects are cached by using only one representative in the computer memory, we save memory usage. On the other hand, if we cache too much, we may use too much memory. To reduce memory usage and costly recomputations by increasing the number of cache hits, i.e. by increasing the quality of caching, objects must be normalized before being kept in the memory or searching for an existing representative. Furthermore, the normalization should be good, i.e. as strong as possible without using too much time.
Technically, for each kind of objects to be cached, TGC2 uses a catalog of objects of that kind, which is either a map or a multimap that uses hash values as keys. Only normalized objects are kept in the catalog. When an object is created not only for temporary usage, for example, to be used as an attribute of another object, we normalize it and search for its representative in the catalog, store it in the catalog if not found, and then return the pointer to the representative for further usages. An object in the catalog is a “hard-resource”, other references to the object are “soft-resources”. We store each object in a catalog together with its usage count. The count is initialized to 1 (for the occurrence of the object in the catalog). When a reference to an object is used as an attribute of another object, the usage count of the former is increased by 1. We release that “soft-resource” by decreasing the usage count by 1. When the usage count of an object is decreased to 1, the object is not used anywhere out of the catalog, and if it is not for permanent caching, the “hard-resource” corresponding to the object will be released (i.e. deleted from the catalog) in the next reduction round of the catalog. Important objects are permanently kept in the catalog, including formulas with the status “satisfiable” or “unsatisfiable” as well as basic instances of the integer linear feasibility problem. Less important objects may also be permanently cached, depending on the used options. Catalog reduction is done periodically, depending on parameters and the effect of the previous reduction. The discussed technique is similar to the techniques of smart pointers of C++ and garbage collection of Java. The differences are that we want to cache objects by using catalogs and some objects are permanently cached.

Another principle of reducing memory usage of TGC2 relies on that we apply dynamic and lazy allocation as well as greedy deallocation for memory management. The types of additional attributes of an object, which may be complex objects themselves, may be known only “on-the-fly”. Furthermore, such attributes may receive values or remain empty. To save memory usage, we allocate memory for an object (or an attribute of an object) only when we really have to, and in a minimal amount, depending on its dynamic type. Apart from that, in most of cases, we release memory used by an object as soon as possible. An exception is reduction of catalogs, which is done periodically. The discussed approach of saving memory increases execution time a bit and makes the code more complicated.

4.3. Search strategies

There are technical problems for which we do not know which from the available solutions is the best. Sometimes, it may be just about selecting a good value for a parameter. Furthermore, different classes of instances of the reasoning problem may prefer different solutions. For this reason, we implement different potentially good solutions for encountered technical problems and parameterize the choices. When everything has neatly been implemented, evaluation will help us in tuning parameters. One of the most important problems is: what search strategy should be used?
Recall that the core reasoning problem is to check whether a given KB is satisfiable. For that, TGC2 constructs a rooted graph which is like an “and–or” graph. Formulas from the labels of the nodes are requirements to be realized and the task is to compute the status of the root, which is either “unsatisfiable” or “satisfiable” or “unsolvable” (caused by interaction between the features \(I, O, Q\) or by exceeding the time limit). In short, the task can be realized as follows: starting from the root, expand the graph, compute and propagate the statuses of the nodes. Propagating known statuses of nodes is done appropriately and has the highest priority. The status of a node may be computed by detecting a direct inconsistency involved with the node (inconsistency means “unsatisfiability” in a certain sense). There are two kinds of inconsistencies: local inconsistency is directly caused by the node and its neighbors, while global inconsistency is caused by impossibility of realizing a requirement with a constructor like existential star modal operators of PDL. When direct detection does not provide a known status for a node, the node may be chosen for expansion, and its status is computed from the statuses of its successors. Mutual dependences between the statuses of nodes may occur. At the end, when all essential computations that may affect the status of the root have been done, if the status of the root is neither “unsatisfiable” nor “unsolvable”, then the status becomes “satisfiable”. Detection of the status “satisfiability” for nodes in a fragment of the graph may be done earlier when it is possible to “close” the fragment.

As discussed above, when no more propagations can be done, we have to choose an unexpanded node and expand it, or choose a node that should be re-expanded and re-expand it (each node may be re-expanded at most once, for dealing with inverse roles or nominals). Re-expansion of a node is done by a specific rule, in a deterministic manner. When an unexpanded node is chosen for expansion, there are choices for how to expand it. The preferences are unary static rules have a higher priority than non-unary static rules, which in turn have a higher priority than transitional rules. A unary (respectively, non-unary) static rule causes the node to become a non-state (an “or”-node) with only one (respectively, more than one) successor. A transitional rule causes the node to become a state. There is in fact only one transitional rule, but we break it into two phases and treat them as two transitional rules, the partial one and the full one (this will be discussed further in the next section).

Which among the applicable unary static rules should be chosen is rather not important. One can imagine that they are applied in a sequence, and the order in the sequence does not really matter. A non-unary static rule realizes a specific requirement in the label of a node (which is called the principal formula of the rule). Which requirement from the label of a node should be chosen for realization first? A general answer is to choose a requirement such that realizing it allows more applications of unary static rules to the successors and the descendants of the node, i.e. so that more simplifications can be done for the successors and the descendants. Among the best candidates (requirements) in respect to this criterion, we choose the “heaviest” one in order to cause the node to have much lighter successors. The weight of a node is
determined by the weight of its label. The weight of a formula is more sophisticated than the size of the formula. It is a subject for optimization and investigation.

For TGC2, we adopt the strategy that, apart from propagations (of various kinds), unary static rules have the highest priority globally. This means that we treat nodes to which some unary static rules are applicable as unstable, and a non-unary static rule or a transitional rule is applied to a node only when all nodes of the graph are stable w.r.t. unary static rules (i.e. no unary static rule is applicable to any node of the graph).

Now, assume that no more propagations can be done and all nodes of the current graph are stable w.r.t. unary static rules. TGC2 chooses a node for expansion or re-expansion. The choice is made by using an expansion queue, which is a priority queue of nodes waiting for expansion or re-expansion. For the priority queue, the reasoner uses a comparison function that depends on attributes of the compared nodes and the used options. Important attributes of nodes and options for determining the priority of a node are the following:

- set-of-supports (SOS) — whether the label of the node contains a formula originated from the query for a query-like checking whether an individual is an instance of a concept w.r.t. a KB or whether a concept is subsumed by another w.r.t. KB, the reasoner checks unsatisfiability of the KB that extends KB with the negation of the query; this technique comes from Otter and other SAT solvers for classical propositional logic;
- the depth of the node in a skeleton tree of the constructed graph: this is used to emphasize depth-first-search (DFS);
- the current limit for the depths of nodes: this is used to emphasize a kind of iterative deepening search; the limit is multiplied by a constant greater than 1 after each deepening round;
- the weight of the label of the node;
- a random number generated for the node.

Each combination of options on how to use the above mentioned attributes provides a search strategy. For example, if the depths of nodes are compared first and the bigger the better, then we have a kind of strong DFS.

In our opinion, strategies based on strong DFS are potentially good as they allow the reasoner to “close” fragments of the graph “on-the-fly” (also note that the graph is finite). However, we do not exclude other strategies. In general, at the current stage of investigation, we use a mixed expansion queue, which mixes different strategies together to obtain a binary tree of priority queues. Each leaf of the tree is a normal priority queue. Each inner node of the tree is labeled by an option and an execution time balance parameter. The left edge outgoing from a node turns on the option, while the right edge turns off the option. The execution time balance parameter specifies the percentage of choosing the left edge out from the two edges to follow. All mixed strategies work on the current graph. For the first version of TGC2, we do
not exploit concurrent computing, but the approach is very suitable for incorporating concurrency.

5. Some Other Optimization Techniques

The earlier version TGC [17] for reasoning in the basic DL $\mathcal{ALC}$ uses a set of optimizations that co-operates very well with global caching and various search strategies on search spaces of the form “and–or” graph. Apart from formula normalization, caching objects and efficient memory management, it also incorporates the following techniques: literal elimination, propagation of unsatisfiability for parent nodes and sibling nodes using “unsat-cores”, and cutoffs. These optimization techniques are very useful for TGC and hence adopted for TGC2.

Some other optimization techniques have already been incorporated into the tableau decision procedures [10, 11, 13, 14] that are used for TGC2, including:

- using finite automata to deal with PDL-like roles;
- using global state caching instead of “pairwise” global state caching for $\mathcal{SHIQ}$ and $\mathcal{SHIO}$; caching nodes in the “local subgraph” of a state in the case of using global state caching;
- using global caching instead of global state caching for $\mathcal{SHOQ}$;
- integer linear feasibility checking for dealing with the feature $Q$;
- techniques for dealing with interaction between the features $I$ and $Q$, between the features $O$ and $Q$, and between the features $I$ and $O$.

TGC2 preprocesses a given TBox $T$ to obtain two parts $T_1$ and $T_2$ such that $T_1$ is an acyclic TBox (consisting of acyclic concept definitions) and $T_2$ is a set of global assumptions. The aim is to make $T_2$ as light as possible. Then, for dealing with $T_1$, TGC2 applies the well-known absorption technique that lazily replaces a defined concept by its definition. To convert and split $T$ to $T_1$ and $T_2$, TGC2 considers different forms of each TBox axiom, different possibilities of grouping TBox axioms, and compares the resulting split TBoxes.

In general, easy tasks are preferred over harder tasks, and the latter are usually delayed in comparison to the former. The strategy adopted for TGC2 is similar. Trying to solve “easy” subtasks first takes less time and this may establish statuses for some nodes of the constructed graph, which may be propagated backward to determine the final status of the root of the graph. TGC2 has options for specifying whether potentially time-consuming subtasks should be delayed. There are two kinds of such subtasks:

- checking “global consistency” for fulfilling eventuality of existential restrictions, which are involved with the PDL-like $*$ role constructor;
- checking feasibility of a set of ILCS, which are related to qualified number restrictions.
As the constructed graph may be exponentially large, checking "global consistency" may be very time-consuming. One can adopt the technique of "on-the-fly" checking for detecting "global inconsistencies" as proposed by Goré and Widmann in [18] for PDL. This probably increases efficiency of the reasoner, as "global inconsistencies" are detected as soon as possible. However, in our opinion, too much more memory is needed, as using this technique each node may contain information about exponentially many nodes in the graph, and this may affect scalability of the reasoner. Our approach for TGC2 is to check "global consistency" periodically.

As strong DFS tries to "close" fragments of the constructed graph and delete them as soon as possible, in combination with strong DFS (which is one among the mixed search strategies) checking "global consistency" may involve only relatively small fragments of the graph.

Checking feasibility of a set of ILCS is NP-hard in the size of the feasibility problem. It makes sense to allow an option for delaying such operations in an appropriate manner. The manner relies on doing partial checks first and delaying full checks. TGC2 uses two kinds of partial checks. A number restriction $\geq n \cdot S.C$ with $n \geq 1$ requires at least satisfaction of $\exists S.C$. So, as the first kind of partial checks, if a number restriction $\geq n \cdot S.C$ belongs to the label of a node, we add the requirement $\exists S.C$ to the label of the node, and we expand a state first only by the transitional partial-expansion rule, which ignores number restrictions (of both kinds $\geq m \cdot R.D$ and $\leq m \cdot R.D$), delaying application of the transitional full-expansion rule, which deals also with number restrictions. As the second kind of partial checks, we relax an integer linear feasibility problem by assuming that the numbers may be real (continuous). It is known that a linear programming problem is solvable in polynomial time.

The current version of TGC2 uses the IBM ILOG CPLEX optimizer and, depending on an option, can use the existing package SCIP (http://scip.zib.de/) for checking feasibility of a set of ILCS.

6. The Current State of TGC2 and a Future Plan

With the intention of making TGC2 competitive with the existing reasoners like RACER or HermiT, which have been optimized over years, we implemented TGC2 very carefully. The development and implementation of TGC2 was very time-consuming, it took us nearly two years and there still remain a lot of extensions, improvements and optimizations to be done. The current version of TGC2 is now available online [9]. Its implementation contains more than 20,000 lines of code in C++, Flex and Bison.

The current version of TGC2 can read a KB in a specific format [9], check its satisifiability and answer queries about instance checking. The part consisting of simple nodes of the tableau constructed for checking the satisifiability of the KB is used through all subsequent processes of answering queries about instance checking. This is an important optimization.
Regarding desired functionalities for TGC2, there remain the following tasks:

- As mentioned earlier, the goal of ontology classification is to compute the hierarchical relation between concepts. Given an ontology specified by a KB (which may be just a TBox), the task is to compute the set of inclusion axioms $A \subseteq B$ such that $A$ and $B$ are atomic concepts (i.e. concept names) and $\text{KB} \models A \subseteq B$. For a specific axiom $A \subseteq B$, checking whether $\text{KB} \models A \subseteq B$ can be done by checking unsatisfiability of the extended $\text{KB}' = \text{KB} \cup \{r : A \cap \neg B\}$, where $r$ is a fresh (new) individual name. Analogously as for instance checking, we can apply a simple but effective optimization technique: the part consisting of simple nodes is common for all of the checks. That is, all of the created simple nodes remain through all of the checks (for different inclusion axioms $A \subseteq B$). Thus, TGC2 avoids costly recomputations. Complex nodes of the constructed graph may be deleted after each check in order to save memory. The classification task is rather simple. We did not realize it because we concentrated on other tasks and we want to find a strategy that reduces the number of inclusion axioms to be checked.

- The next functionality that should be added to TGC2 is to allow inputs in the OWL format. Using Flex and Bison, the task should not be a big problem. Reading a RIA-automaton-specification also remains to be implemented.

- Recall that, for the current version of TGC2, the PDL-like $^*$ role constructor can be used only when the feature $O$ is absent and either the feature $Q$ or the feature $I$ is absent. We will try to extend TGC2 so that the restriction is reduced to “there are no interactions between the features $I$, $O$ and $Q$”.

- The extension with functionality, irreflexivity and disjointness of roles, the universal role, the concept constructor $\exists r.\text{Self}$, data roles (properties), basic predicates on data (numbers, dates and strings) can be done for TGC2 without difficulties w.r.t. theoretical foundation.

- An interface compatible with OWL API or DIG is needed for TGC2 to allow it to be used in other systems.

- Returning an explanation for the answer of a query (in the form of an unsatisfiability core, or an interpretation in the case of satisfiability) remains to be done for TGC2.

Regarding improvements and further optimizations for TGC2, there remain the following tasks, which may require about 10,000 more lines of code and one year for realization:

- Computing and propagating “unsat-cores” is a very useful optimization of TGC [17]. For the rich DL supported by TGC2, the task is complicated and was not yet implemented. It remains to be done for TGC2.

- Splitting a given ABox into independent parts in order to reduce the main task to smaller independent subtasks is very important when the ABox is large. That is,
“divide and conquer” is a very important strategy (especially, as we have here an \textsc{ExpTime}-complete problem). Furthermore, simulating successful reasoning of previous subtasks for a next “similar” subtask is also a promising approach. Such optimizations are planned for TGC2.

- Global assumptions from the TBox are included in the label of each node after a transition in the constructed tableau. We intend to do a better preprocessing for them and compact them in order to increase efficiency and reduce memory usage of the reasoner. This would be useful for large TBoxes.
- Tuning parameters for TGC2 remains to be done.

7. Preliminary Evaluation

A considerable number of tests have been done for TGC2 [9]. Their aim was mainly to detect and eliminate bugs. We did not conduct a comprehensive evaluation of TGC2 using large ontologies, as there remain important intended optimizations to be implemented. However, to justify the usefulness of our tableau method for the DL \textsc{SHIQ} [10], we have compared TGC2 with the existing well-known reasoners RACER, HermiT, FaCT++, Pellet and KAON2 w.r.t. dealing with qualified number restrictions by using the following examples. In each of the examples, we have a KB in \textsc{SHIQ} and the task is to check whether KB is satisfiable.

**Example 1.** The considered KB consists of only the following assertions, where \( n \) is a constant natural number (e.g. 500 or 1000):

\[
\begin{align*}
    a & : \forall r. (\neg A \sqcup \neg B), & a & : \geq n \cdot r. A, & a & : \geq (n + 1) \cdot r. B, & a & : \leq 2n \cdot r. (A \sqcup B).
\end{align*}
\]

**Example 2.** The considered KB consists of the following axioms and assertions, where \( n \) is a constant natural number (e.g. 1, 5 or 100,000) and the first four axioms together state that \( A, B, C \) and \( D \) are pairwise disjoint concepts:

\[
\begin{align*}
    A & \sqsubseteq \neg (B \sqcup C \sqcup D), & (1) & \\
    B & \sqsubseteq \neg (A \sqcup C \sqcup D), & (2) & \\
    C & \sqsubseteq \neg (A \sqcup B \sqcup D), & (3) & \\
    D & \sqsubseteq \neg (A \sqcup B \sqcup C), & (4) & \\
    E & \sqsubseteq \leq (4n + 2) \cdot r. (A \sqcup B \sqcup C \sqcup D), & (5) & \\
    a & : E, & (6) & \\
    a & : \geq (2n + 1) \cdot r. (A \sqcup B), & (7) & \\
    a & : \geq (2n + 1) \cdot r. (A \sqcup C), & (8) & \\
    a & : \geq (2n + 1) \cdot r. (A \sqcup D), & (9) & \\
    a & : \geq (2n + 1) \cdot r. (B \sqcup C), & (10) & \\
    a & : \geq (2n + 1) \cdot r. (B \sqcup D), & (11) & \\
    a & : \geq (2n + 1) \cdot r. (C \sqcup D). & (12) & \\
\end{align*}
\]
Example 3. The considered KB differs from the one in the previous example only in that the axiom (6) is replaced by the following assertion and role axioms (which state that the inverse of r is a subrole of s and s is transitive):

\[
\begin{align*}
\alpha : & \exists r. \exists r. \forall s. E, \\
\bar{r} & \subseteq s, \\
s \circ s & \subseteq s.
\end{align*}
\]

Example 4. The considered KB differs from the one in Example 3 in that \(4n + 2\) in (5) is replaced by \(4n\), and the occurrences of \(2n + 1\) in (7)–(12) are replaced by \(2n\).

Observe that the KBs in Examples 1–3 are unsatisfiable, while the one in Example 4 is satisfiable. We have tested the well-known reasoners RACER (2.0), HermiT (1.3.8), FaCT++ (1.6.4), Pellet (2.2.0), KAON2 (2008-06-29) and our reasoner TGC2 (006.beta) by using the above examples. The tests were conducted by using a PC with the following parameters: Intel® Xeon(R) CPU E3-1220 v3 @3.10 GHz × 4, 8 GB, Ubuntu 14.04.3 LTS 64-bit, JRE 1.7. The reasoners RACER, HermiT, FaCT++ and Pellet were tested using Protégé 4.3 (by calling the command “Start reasoner”), KAON2 was tested using Protégé 3.5 (by using DIG and calling the command “Check consistency”), and RACER was tested by using its server and RacerPorter.

The test results are presented in Table 1. As KAON2 (2008-06-29) via Protégé 3.5 could not solve Example 1 with \(n = 500\) (it reported ArrayIndex Out Of Bounds-Exception) and did not find the inconsistency in Examples 2 and 3 with \(n = 1\), we do not include its test results into Table 1. The command “Start reasoner” of Protégé 4.3 first checks whether the considered KB is satisfiable, and if it is, then the program executes some additional tasks. Thus, the reported amounts of consumed resource (in the cases related with Protégé) for Example 4 should be understood loosely. Besides, the reported amounts of consumed memory in the case of HermiT, FaCT++ and Pellet contain the amount of memory consumed by Protégé 4.3 (about 175 MB).

Observe that RACER (2.0) deals very well with Examples 1 and 2. It recognizes the logic used in Example 1 as \(\mathcal{ALCQ}\) and the logic used in Example 2 as \(\mathcal{ALCTQ}\). For Examples 3 and 4, recognized as examples in \(\mathcal{SHIQ}\), RACER (2.0) performs worse than HermiT (1.3.8), at least for the case \(n = 1\). Also observe that FaCT++ (1.6.4) tends to use too much memory and HermiT (1.3.8) cannot deal with big numbers.

As can be seen from the test results in Table 1, the existing reasoners Pellet (2.2.0), FaCT++ (1.6.4), HermiT (1.3.8) and RACER (2.0) are not scalable w.r.t. qualified number restrictions in \(\mathcal{SHIQ}\) and are far from being satisfactory.

b The input files in the text format for TGC2 and the OWL format for the other reasoners are available at http://mimuw.edu.pl/~nguyen/TGC2/TESTS.
Table 1. Test results for reasoners in DLs.

<table>
<thead>
<tr>
<th>Exp., n</th>
<th>Pellet (2.2.0)</th>
<th>FaCT++ (1.6.4)</th>
<th>HermiT (1.3.8)</th>
<th>RACER (2.0)</th>
<th>TGC2 (006.beta)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Y: \approx 1.75 min, \approx 254 MB</td>
<td>N: (H1)</td>
<td>Y: \leq 1 s, \leq 31 MB</td>
<td>Y: \leq 6 ms, \approx 1.0 MB</td>
<td></td>
</tr>
<tr>
<td>1, 500</td>
<td>N: (F1)</td>
<td>Y: \approx 1.75 min, \approx 254 MB</td>
<td>N: (H1)</td>
<td>Y: \leq 1 s, \leq 31 MB</td>
<td>Y: \leq 6 ms, \approx 1.0 MB</td>
</tr>
<tr>
<td>1, 10^4</td>
<td>N: see above</td>
<td>Y: \approx 27.5 min, \approx 310 MB</td>
<td>N: (H2)</td>
<td>Y: \leq 1 s, \leq 31 MB</td>
<td>Y: \leq 6 ms, \approx 1.0 MB</td>
</tr>
<tr>
<td>2, 1</td>
<td>U: &gt; 39 h, &gt; 447 MB</td>
<td>N: &gt; 6.9 GB, (F1)</td>
<td>Y: \approx 26 min, \approx 403 MB</td>
<td>Y: \leq 1 s, \leq 31 MB</td>
<td>Y: \leq 6 ms, \approx 20.5 MB</td>
</tr>
<tr>
<td>2, 5</td>
<td>U: see above</td>
<td>N: see above</td>
<td>U: &gt; 46 h, &gt; 228 MB</td>
<td>Y: \leq 1 s, \leq 31 MB</td>
<td>Y: \leq 6 ms, \approx 20.5 MB</td>
</tr>
<tr>
<td>2, 10^4</td>
<td>N: (PH)</td>
<td>N: see above</td>
<td>N: (PH)</td>
<td>Y: \leq 1 s, \leq 31 MB</td>
<td>Y: \leq 6 ms, \approx 20.5 MB</td>
</tr>
<tr>
<td>3, 1</td>
<td>U: &gt; 46 h, &gt; 448 MB</td>
<td>N: &gt; 7.3 GB, (F1)</td>
<td>Y: 24-25 min, \approx 363 MB</td>
<td>Y: 10-26 h, \approx 2.2 GB</td>
<td>Y: 9.29-9.69 s, 69-74 MB</td>
</tr>
<tr>
<td>3, 5</td>
<td>U: see above</td>
<td>N: see above</td>
<td>U: &gt; 40 h, &gt; 213 MB</td>
<td>U: &gt; 15 days, &gt; 5.7 GB</td>
<td>Y: 9.29-9.69 s, 69-74 MB</td>
</tr>
<tr>
<td>3, 10^4</td>
<td>N: (PH)</td>
<td>N: see above</td>
<td>N: (PH)</td>
<td>N: (H1)</td>
<td>Y: 9.29-9.69 s, 69-74 MB</td>
</tr>
<tr>
<td>4, 1</td>
<td>Y: \approx 4.9 min, \approx 338 MB</td>
<td>Y: \approx 30 ms, \approx 179 MB</td>
<td>Y: \approx 1 s, \approx 183 MB</td>
<td>Y: \approx 6 s, \approx 59 MB</td>
<td>Y: 0.79-0.95 s, 32-33 MB</td>
</tr>
<tr>
<td>4, 5</td>
<td>U: &gt; 39 h, &gt; 470 MB</td>
<td>U: &gt; 7.2 GB, &gt; 0.5 h, (F2)</td>
<td>U: &gt; 40 h, &gt; 199 MB</td>
<td>U: &gt; 21 days, &gt; 6.4 GB</td>
<td>Y: 0.79-0.95 s, 32-33 MB</td>
</tr>
<tr>
<td>4, 10^4</td>
<td>N: (PH)</td>
<td>N: &gt; 7.2 GB, (F3)</td>
<td>N: (PH)</td>
<td>N: (R1)</td>
<td>Y: 0.79-0.95 s, 32-33 MB</td>
</tr>
</tbody>
</table>

Notes: The letter Y (respectively, N) means the considered problem can (respectively, cannot) be solved by the considered reasoner (using our PC). The letter U means we do not know whether the considered problem can be solved by the considered reasoner (i.e. we did not wait long enough for a result). The texts in brackets are references to the remarks given in the below part of the table.

P1 “OutOfMemoryError: GC overhead limit exceeded” in 0.5-4.0 h.
P2 “OutOfMemoryError: GC overhead limit exceeded” in the first minute.
F1 Killed by the operating system in the 6th minute.
F2 The reasoner used more than 7.2 GB already in the 18th minute.
F3 Not enough memory (the reasoner used more than 7.2 GB already in the second minute).
H1 “OutOfMemoryError: Java Heap Space” in the 6th minute.
H2 “OutOfMemoryError: Java Heap Space” in the 4th minute.
R1 Not enough memory (the reasoner used more than 7.0 GB already in the 10th minute).
The test results in Table 1 show that TGC2 deals with qualified number restrictions in $\text{SHIQ}$ much better than the other existing reasoners w.r.t. both efficiency and memory usage. The point is not that the performance of TGC2 is a few orders of magnitude better than the other existing reasoners for Example 3 with $n = 1$. The difference in performance for such examples is rather a matter of an exponential (or double exponential) function of $n$. The examples used in the test are not anything specially unusual. They are the first ones we had for experimenting with $\text{SHIQ}$. Example 3 is a simple example with only five concept names, two role names, two role axioms, five terminological axioms and seven individual assertions. When $n = 1$, the numbers occurring in the example are not greater than six.

8. Conclusions

The theoretical foundation of TGC2 is our tableau methods with $\text{ExpTime}$ (optimal) complexity developed recently for CPDL$_\text{reg}$ [13] and the DLs $\text{SHIQ}$ [10], $\text{SHOIQ}$ [11] and $\text{SHIO}$ [14]. They are significant theoretical results, but too complicated and thus not included into this paper.

In this work, we have presented the design of TGC2, providing the design principles of TGC2 and optimization techniques that have been developed and implemented for TGC2. We also discussed further optimization techniques that we intend to implement for TGC2 in the near future. All of these principles and optimization techniques have been carefully analyzed and chosen in order to guarantee high efficiency.

The implementation of TGC2 was very time-consuming, and as some important intended optimizations were not yet implemented, we did not conduct a comprehensive evaluation for TGC2. We have done, however, a preliminary evaluation for TGC2 w.r.t. dealing with qualified number restrictions. The experimental results showed that TGC2 deals with qualified number restrictions in $\text{SHIQ}$ much better than the well-known reasoners like RACER and HermiT, which have been optimized over years. The reason is simple: our technique is advanced and global caching guarantees an optimal complexity for $\text{ExpTime}$ modal and DLs. The latter distinguishes TGC2 from the existing tableau reasoners for DLs.

There remain certain tasks to do for extending and improving TGC2. From our experience with TGC [17], we believe that after doing those tasks, the new version of TGC2 will be an efficient reasoner for modal and DLs.

References