Bachelor Thesis
Institut für Informatik der Freien Universität Berlin

Work Group: Artificial Intelligence

A framework for inconsistency detection in expressive ontologies

Marco Ziener
Matrikelnummer: 4359973
marco.ziener@fu-berlin.de

Primary Reviewer and advisor: PD Dr. Christoph Benzmüller
Secondary Reviewer: Prof. Dr. Marcel Kyas

Berlin, June 25, 2013

Abstract

Expressive ontologies provide a useful tool for the precise and short description of knowledge at increased costs for computation. Due to the expressiveness it is also easier to introduce contradictions and inconsistencies into the ontology and thus disallow to apply it in practical problems. In this work an extendable framework for the detection of inconsistencies is described and implemented on the basis of a graph representation of the ontology. The framework is used concretely for checking the partial consistency of the Suggested Upper Merged Ontology. In the end different kinds of errors are categorized and analyzed.

Eidesstattliche Erklärung

Ich versichere hiermit an Eides Statt, dass diese Arbeit von niemand anderem als meiner Person verfasst worden ist. Alle verwendeten Hilfsmittel wie Berichte, Bücher, Internetseiten oder ähnliches sind im Literaturverzeichnis angegeben, Zitate aus fremden Arbeiten sind als solche kenntlich gemacht. Die Arbeit wurde bisher in gleicher oder ähnlicher Form keiner anderen Prüfungskommission vorgelegt und auch nicht veröffentlicht.

Marco Ziener	Datum

Contents

1 Introduction

In a big part of the 20th century higher-order logic was a rather theoretic topic lacking practical application due to the lack of computational power. But since computational power has significantly increased it became feasible to employ it in different problem domains. It excels when knowledge is formalized due to the duality of being human readable as well as being easily interpreted by a machine. This duality provides great advantages when it comes to mechanical work with defintions and knowledge allowing the human driver of the system to excel in the domain of heuristic reasoning. Additionally, because of the uniform knowledge representation it can be translated into different languages and thus put into greater use as well as ease the scientific discourse.
Once ontologies belonged to the realm of philosophy giving the philosopher the chance to think about the totality of existing entities and their shared properties[1]. However with the advent of artificial intelligence the term ontology is coined differently: It has become a representation vocabulary which is not defined by the meaning of the symbols but by the emergent interconnections between the symbols. Thus an ontology becomes representation independent since the relations are preserved through mere renaming of the symbols while avoiding name space collisions. By constructing an ontology for a specific problem domain one captures the conceptual structure of the domain and - given some sort of representation - can communicate his knowledge and discuss it with others more easily. It becomes natural to combine higher-order logic with its expressiveness as a language for the efficient representation of ontologies. This key idea forms the notion of an expressive ontology like the Suggested Upper Merged Ontology (SUMO).
However, these desirable properties come with a price: the computational effort for automated reasoning and the sheer amount of formalized knowledge hinder to guarantee and check the total correctness and consistency of the ontology and thus lead to a flawed reasoning process or wrong categorizations of new knowledge. Furthermore when introducing new axioms contradictions and errors can be easily created due to the sheer size of the ontology and the complex interactions of the axioms.
At the same time checking for inconsistencies is a rather tedious and time consuming job, since the knowledge database is usually rather large and a lot of computational effort must be invested. On the level of first-order logic several different approaches for inconsistency detection have already been made[2] in SUMO. The work done on inconsistency detection in higher-order logic is sparse[3] even though it is highly probable that there are several mistakes undetectable by

[1] Most prominently in the 20th century: [1]. However metaphysics and ontology are a focalpoint since the greek philosophers.

[2] Most recently:[2], but also before: e.g. [3] and others. The recent findings suggest that there are still mistakes to be found.

[3] For instance: [4]

first-order logic due to the lack of types. Due to subsuming several other logics like epistemic, doxastic and temporal and not giving them special representations - and thus special semantics - it can be the case that their special semantics are broken and they are rendered useless and wrong. Even though axioms using them can be satisfiable but turn out to be wrong when asked to derive a conjecture with them. An example for such behavior will be given after introducing the necessary notions.
All in all this thesis implements a framework for the mechanized search for inconsistencies in expressive ontologies and uses it to search for these inconsistencies in SUMO although the framework can be applied to any expressive ontology which is written in the THF0 file format.

2 Fundamentals

2.1 Suggested Upper Merged Ontology (SUMO)

The Suggested Upper Merged Ontology [5] is an open source formal *upper ontology*[4]. Developed by Teknowledge Corporation and now maintained by Articulate Software it is a candidate for the *standard upper ontology* of the IEEE working group 1600.1. During the 10 years of development it was reviewed by many people and found application in automatic theorem proving and reasoning. In order to distinguish between different levels of abstraction SUMO is split into a three layer architecture. As an upper ontology it contains at the core, knowledge about the general notions and concepts of set/class theory, numeric [5] and temporal aspects of reasoning such as processes and events. Upon these notions the mid layer - also known as MILO - is defined which serves as a bridge between the abstraction of the core level and the concrete knowledge of several domains. The outer layer contains the domain specific knowledge defined upon the notions of the inner layers. On the outer layer several different fields of knowledge are described. For instance knowledge about weapons of mass destruction, physical elements, media and distributed computation is described. The diversity and thus the broad coverage of the knowledge was achieved by mapping the entire WordNet lexicon to SUMO [6]. This even allows SUMO to be used as a basis for natural language understanding tasks. It also contains language generation templates for Hindi, Chinese and English among others allowing to restate knowledge automatically in different languages. If all the domains are combined, SUMO contains about 70.000 axioms and around 20.000 terms.
SUMO is written in SUO-KIF, a simplified version of the Knowledge Interchange Format (KIF) [7]. However, it is also available in Web Ontology Language (OWL) and can be translated using the SIGMA [8] system for formal ontology development into the *THF0 language* [9] for classical higher-order logic. In this translation it is regularly used as a playground and benchmark for the proving capabilities of several automated theorem provers.

2.2 Brief introduction to higher-order logic

2.2.1 A Brief History of Higher-Order Logic

The study of reasoning has a very long history over several different countries and cultures. Meanwhile several principles were discovered and described earlier[6], the advent of modern logic began in the 19th century with the publication of

[4] As such it describes general concepts over many specific domains as well as domain specific knowledge.

[5] And thus graph and measurements.

[6] See the works of Leibniz, Bolzano and Hobbes. However the ideas were isolated and the authors failed to formulate a comprehensive system of their ideas.

Boole's *The Mathematical Analysis of Logic* [10] and DeMorgan's *Formal Logic* [11] in 1847. Both books introduced the idea that logical relationships could be represented in an algebraic system and created a whole movement which explored several different aspects of this idea. But even then mechanization of logic was of central interest. For instance mechanization in form of a *logical machine* for this type of logic was realized in 1870 [12, p. 405]. After several defects of Boole's original system were fixed and it was simplified by the introduction of the *exclusive or*, the limited expressivity of the logic showed: the concept of relationships was not clearly defined and there was no discrimination between quantified expressions and singular statements. This issue was later fixed by Frege's *Begriffsschrift* [13] in 1879 and led to a movement to provide mathematics with a formal foundation known as the *mathematical school*[7]. According to Collins [14] a theory of types was already anticipated by Schröder [15] in 1890 even though he failed to formulate it exactly. Meanwhile the *Begriffsschrift* was rigorous in its axioms and approach to logic Russel found in 1901 the *Russel antinomy*, which showed that the added expressivity of Frege's system led to a contradiction and even to a vicious circle. Although several other similar paradoxes[8] were known, its publication [16] in 1903/04 shattered the axioms of existing set theory. Concerned with the correctness of mathematics he also proposed a solution to the issue in the appendix: the introduction of a theory of types. This lead up to his work with Whitehead [17] where both of them tried to derive all mathematical truth by introducing a type system to limit the expressivity and thus avoid the paradoxes he earlier discovered. Until 1931 several different approaches and ideas were checked for a theory of types. Most influential were the works of Chwistek [18] and Ramsey [19] which suggested that the hierarchy on types as constructed by Russel could be collapsed. However only in 1931 the first version of a simple theory of types was presented by Gödel [20] and Tarski [21] independently. Today's simple type theory was introduced by Church [22] with his developed λ-calculus as a basis. Several of his students[9] have continued his work on simple type theory and provided it with well studied semantics and applications. Due to the theoretical nature of higher-order logic it has been neglected for the last decades but is currently gaining popularity since several automated theorem provers were developed. Additionally a common platform for automated theorem provers in form of Ten Thousand Solutions for Theorem Provers (TSTP) and Ten Thousand Problems for Theorem provers (TPTP) has lead to standardization between different systems and allows the public to use expert systems for reasoning without setting up the system on the local machine. By introducing higher-order logic to this platform [23] it can be used freely alongside the well known first-order logic automated theorem provers by the general public and gain more popularity.

[7] It consisted of Hilbert, Dedekind and several others.

[8] E.g. Burali-Forti paradoxn, Cantor's paradox, etc.

[9] Especially Andrews and Henkin.

2.2.2 Simple type theory

Simple type theory [10] is a natural extension of the well known first-order logic. Nevertheless it is not the only framework to express higher-order logic, but it excels with its practical properties. However where first-order logic shies away from terms denoting higher-order objects like sets and applying predicates and functions to those terms in fear of contradictions, simple type theory uses types to limit the expressiveness and provide solid ground for reasoning. This approach provides several advantages:

- It eases the formulation and communication of knowledge due to its expressiveness.
- It allows to formulate specification for large systems and thus aid construction, verification and validation of these systems.
- It provides often an elegant way to express properties in contrast to predicate logic.

This section provides an overview about the most relevant terms used in the framework and their relationships. It was adapted from various sources mainly [24, 25, 26, 27]. Please consider the reference therein for further exposure of the topic. Thus the overview is not intended to be complete. However since the best studied and most used semantics for **STT** are Henkin semantics they will be used without further ado.
STT is built up from 2 different *types* of objects: *types* and *expressions*. **type[α]** assumes that α is a type meanwhile **expr[E,α]** assumes that E is an expression of type α. The set of all types **T** of **STT** consists of 3 different types: $\iota, o, \rightarrow$.

Type of individuals

$$\frac{}{\textbf{type}[\iota]}$$

The type symbol ι denotes the type of individual values.

Type of truth values

$$\frac{}{\textbf{type}[o]}$$

The type symbol o denotes the type of truth values.

Function type

$$\frac{\textbf{type}[\alpha], \textbf{type}[\beta]}{\textbf{type}[\alpha \rightarrow \beta]}$$

If $\alpha, \beta \in \mathbf{T}$ then the arrow creates a function type between them. This allows the creation of compound types out of *basic* types.

Let **V** denote the set of all possible variables. In order to construct expressions a few syntactical symbols must be introduced. For the sake of comprehensiveness

[10] In the following text it will be abbreviated by **STT**.

they are introduced with the application in the calculus. Later this will prove handy when describing the THF0 file format in which the expressive ontology is represented.

Type binding (:)

$$\frac{x \in \mathbf{V}, \mathbf{type}[\alpha]}{\mathbf{expr}[(x : \alpha), \alpha]}$$

With this operator a variable $x \in \mathbf{V}$ can be bound to a type $t \in \mathbf{T}$.

Function abstraction (λ)

$$\frac{x \in \mathbf{V}, \mathbf{type}[\alpha], \mathbf{expr}[B, \beta]}{\mathbf{expr}[\lambda x : \alpha, \alpha \to \beta]}$$

Similar to the λ-calculus this allows to declare a λ-function which for instance can be used to declare relations between types.

Function application (@)

$$\frac{\mathbf{expr}[A, \alpha], \mathbf{expr}[F, (\alpha \to \beta)]}{\mathbf{expr}[(F@A), \beta]}$$

Given an expression A of type α and an expression F of the function type $\alpha \to \beta$ a new expression $(F@A)$ of type β can be created.

Equality (=)

$$\frac{\mathbf{expr}[E_1, \alpha], \mathbf{expr}[E_2, \alpha]}{\mathbf{expr}[E_1 = E_2, o]}$$

With this operator the equality for types is introduced.

Definite description (I)

$$\frac{x \in \mathbf{V}, \mathbf{type}[\alpha], \mathbf{expr}[A, o]}{\mathbf{expr}[I x : \alpha . A, \alpha]}$$

The value of the definite description is the unique value of x if it can satisfy condition A else a canonical error element of that type will be returned.

With this formation rules the language $\mathbf{L} = (\mathbf{C}, \tau)$ can be defined where $\mathbf{C}$ is the set[11] of all constants and $\tau : \mathbf{C} \to \mathbf{T}$. Any formula in $\mathbf{L}$ has the type o. Parentheses and types will be omitted when meaning is not lost. Types may be written as subscript. The notions of *free* variables and *bound* variables carry directly over from first-order logic. Without further ado it is assumed that enough special constants for connective declaration exist and that those can be defined by $\neg_{o \to o}, \vee_{o \to o \to}$, and $\Pi_{(\alpha \to o) \to o}$ with the intuitive semantics. In general the base types ι and o are not enough and the base types are extended by user defined types. This mechanism is extensively used by SIGMA when translating to a **STT**-representation from a first-order representation by inferring types from the formulas. For further discussion it is useful to determine how the notions of validity and satisfiability are lifted from first-order logic. However the semantics of higher-order logic are different and more complicated than first-order logic.

[11] This set is disjoint with $\mathbf{V}$.

For this purpose the notion of a universe must be introduced first. A universe **U** is a collection of sets with the following properties:

- Every set of **U** is nonempty;
- If a set is in **U** then also all of its subsets are in **U** but the empty set;
- For every $S_1, S_2 \in \mathbf{U}$ also $S_1 \times S_2 \in \mathbf{U}$;
- For every set S in **U** also $P(S)$ must be in **U**;
- **U** contains a marked infinite set **I**;
- $A \mapsto B$ is the set of all functions from A to B. From the previous characterizations it follows that $A \mapsto B \subset P(A \times B)$ and $A, B \in \mathbf{U} \implies A \mapsto B$.

With this notion it is possible to introduce the notion of a frame. A frame is a collection $\{D_\alpha\}_{\alpha\in\tau}$ with $D_\alpha \in \mathbf{U}$ for $\alpha \in \tau$ and the following distinct domains: $D_o = T, F$, D_ι = an infinite set of individuals; $D_{\alpha\mapsto\beta}$ = some collection of functions from D_α to D_β. An interpretation for an formula can now be defined by a pair $< \{D_\alpha\}_{\alpha\in\tau}, \mathbf{I} >$ where $\{D_\alpha\}_{\alpha\in\tau}$ is a frame and **I** is an interpretation function which maps each typed constant to an element from the corresponding typed domain. It obeys a certain rules:

- $\mathbf{I}(T) = T$ and $\mathbf{I}(F) = F$;
- $\mathbf{I}(=_{\alpha\to\alpha\to o})$ serves as identity on D_a;
- Negation and disjunction have the intuitive semantics on D_o;
- $\mathbf{I}(I_{(\alpha\to o)\to\alpha}) \in (D_\alpha \to D_o) \mapsto D_a$ describes the function which if and only if there is a unique element in D_α which satisfies a condition specified by the function returns it. Otherwise it returns an arbitrary element.

Let σ be a substitution similar to first-order logic. A model for higher-order logic is described by $M =< \{D_\alpha\}_{\alpha\in\mathbf{T}}, \mathbf{I} >$ if and only if there exists a binary function V^M with the properties for all type-indexed families of substitutions $\sigma = (\sigma_\alpha)_{\alpha\in t}$ and terms t of type α, $V^M(\sigma, t) \in D_\alpha$

- $V^M(\sigma, x_\alpha) = \sigma_\alpha(x_\alpha)$
- $\forall c \in \mathbf{C} : V(\sigma, c) = \mathbf{I}(c)$
- $V^M(\sigma, s_{\alpha\mapsto\beta} t_\alpha) = V^M(\sigma, s) V^M(\sigma, t)$
- $V^M(\lambda x_\alpha . t_\beta) = \forall z \in D_\alpha (D_\alpha \mapsto D_\beta)(z) = V^M(\sigma[x \leftarrow z], t)$

This gives rise to the notions of satisfiability and validity: A formula ϕ is satisfiable in M if $V^M(\sigma\phi) = T$ for some substitution σ. A formula ϕ is valid in M if $V^M(\sigma\phi) = T$ for every substitution σ. A formula is generally $valid$ if it is valid in every general model M. With Henkin semantics **STT** shares to important properties with first-order logic: it is semi-decidable and compact.
Proof procedures for this expressive logic are difficult since it subsumes several *lesser* logics whose proof procedures are in general already a difficult topic of their own. Therefore proof procedures for higher-order logic will not be covered here. The same issues apply to the algorithms for model generation.

2.2.3 The THF0-format for simple type theory

THF0 [28] is a *conservative extension* to the language used at the TSTP/TPTP platform. It lifts the earlier defined untyped first-order to typed higher-order logic in form of **STT** . In general a formula is given by

```
thf(name,role,formula[optional annotations]).
```

where

name An unique identifier for the formula denoted in the expression.

role Most commonly this is either *axiom* or *conjecture*. Since this work focuses on the work with ontologies usually *axiom* is found at this position. By putting *conjecture* at this position the user request the automatic theorem prover to prove the statement given the earlier defined axioms. Also it is possible to introduce new types into the system by using the keyword *type* here. For more extensive discussion of the various (and even more) roles consider the aforementioned paper.

formula A **STT**-formula in appropriate THF0 representation.

Above discussion left out what appropriate representation for formulas is. The standard types of **STT** are mapped to \$$i$ (ι),\$$o$ (o) and > ($\rightarrow$). Type declarations take the form $symbol : signature$, which corresponds to the earlier formulated rule of typing a variable. As in many programming languages constant and variable type symbols must be declared before they are used in axioms. The *usual* quantifiers and connectives are lifted form the first-order form of the TPTP syntax. Therefore the following mapping ($\leftrightarrow$) is used to express the classical operators of logic:

$$? \leftrightarrow \forall$$

$$! \leftrightarrow \exists$$

$$\sim \leftrightarrow \neg$$

$$\& \leftrightarrow \wedge$$

$$| \leftrightarrow \vee$$

$$=> \leftrightarrow (\Rightarrow)$$

$$<= \leftrightarrow \Leftarrow$$

$$<=> \leftrightarrow \Leftrightarrow$$

$$<\sim> \leftrightarrow \oplus$$

Besides this expression of common logical operators, THF0 does not assume any special semantics concerning the higher-order logic it expresses. This allows the user to choose the semantics appropriate to his needs. The downside is that there are a lot of expression which are legal in THF0 but are meaningless. For validation of the types a external system like Twelf [29] must be employed. THF0 does not support polymorphism for the symbols declared and used. To sidestep this restriction the SIGMA translation algorithm uses semantic identical symbols and appends a different suffix with the type encoded as way to *overload* the semantic symbols and create some sort of polymorphism. On several occasions it shows that several symbols used in SUMO do not follow consistent usage in first-order logic. As it will be described it is unfortunate that the annotations for formulas are so rarely used and the user has to search mistakes in SUMO manually. Also it makes the validation of the translation of SIGMA difficult since one has rely on the correctness of the algorithm and the correctness of the implementation.

3 The framework

3.1 General considerations concerning the framework

There were several considerations before the implementation of the framework that lead to the design decisions in the framework.

Reuse Albeit the framework is implemented for SUMO, it should be possible to use it for its purpose on other ontologies. Therefore most of the parts must be written in generic fashion leaving the details to the user. Meanwhile in SUMO all of the types can be extracted by regular expressions, this may not be the case with other ontologies. However with the chosen programming language it should be relative to change certain aspects of the framework and employ new combinations of the modules.

Flexibility Not all execution environments are created equal and not all deployments offer the same technical possibilities. The deployment of the framework should take this into consideration and allow the user to cater it towards his individual needs. Furthermore it cannot be assumed that the data and its meta information remain static. If they are extended there must be a way to work with the newly acquired data. With the generation of data the data storage should be able to adapt and thus be able to add additional systems for this purpose. Furthermore, even similar execution environments can have varying costs for operations and therefore it can be useful to implement operations slightly different and leave costly operations out.

Extensible The current approach to inconsistency detection is rather strenuous for the computer. There exists the possibility that new algorithms for this purpose can be created and that their efficiency is much greater than the current - rather naive - approach. The framework must pay tribute to the technical development and allow them to be integrated into the framework. Therefore it should provide the user with practical abstractions of the ontology to allow the creation of such algorithms. At the same time it should allow to add new - perhaps more powerful - provers to the system and be able to use them on the ontology.

Report It should be possible to infer information about certain properties of the ontology like the amount of interconnections of an axiom and judge the quality of the given part of ontology. For instance it would be highly desirable to find out which axioms are most general and refactor them into the higher levels of the ontology.

Resource consumption The detection of inconsistencies is quite a intensive job - as well when it comes to memory consumption as well as the necessary computation power. Meanwhile the necessary computational resources are

a necessity inherent to the problem, memory consumption is not and must be reduced as much as possible since power sets tend to get quite large.

3.2 Used technology

Python Python[12] is 4th generation high level programming language developed by Guido van Rossum and supports several different programming paradigms. It is backed by a large community and has excellent library support ranging over several different domains. Python has a low amount of keywords and allows the user to work on a high level of abstraction by supporting functions as first-class objects. Furthermore Python facilitates the programming of *glue-code* [13]. Together with an interactive interpreter and the platform independence these properties make Python a natural choice as a programming language for the framework.

MongoDB MongoDB[14] is a high-performance open source database. It is developed by 10gen and used by SAP, craigslist, Sourceforge, wordnik and several others[15]. In contrast to classical SQL-based databases it is schemaless and document-oriented. In MongoDB data is stored in BSON-format[16] which - together with the good language support for several programming languages - allows the easy integration of the data into the application. Data can be queried by field, range query or regular expression. Also it is possible to index data as in more standard SQL-databases by any field. For huge amounts of data it can become convenient to make use of *sharding*[17] capabilities as well as the MapReduce possibilities of MongoDB. The considered alternatives lacked in general certain technical features[18], were not so easy to integrate into the application or contradicted the general considerations beforehand. Additionally applications developed with MongoDB are in general driven by relations in the program logic rather than a given schema. Since the framework should leverage the user to work with data by his own ideas and algorithms this a highly desired feature.

Celery Celery[19] is an open source Python library implementing queues for job submission by message passing. It offers several different backends for this purpose. One of them is RabbitMQ which was chosen for the backend of Celery. RabbitMQ is open source message broker implementing

[12] www.python.org

[13] The term glue-code describes code that *glues* several different independent programs together and allows the creation of compound programs.

[14] http://www.mongodb.org

[15] See: http://www.mongodb.org/about/production-deployments/

[16] Essentially a binary relative of the well-known JSON-format.

[17] Therefore the horizontal partition of the database onto several smaller systems.

[18] In general sharding and concurrency are difficult to find.

[19] http://http://www.celeryproject.org/

the advanced message queuing protocol [30]. It is built upon the Open Telecom Platform (OTP) and provides very good properties when it comes to clustering and failover. These two properties can be exploited to great effect when the framework is deployed onto a cluster.

TORQUE The TORQUE Resource manager [31] is a cluster resource manager based on the Portable Batch System provided under an open source license. It allows the user to distribute his jobs onto a cluster of different machine and specify fine grained control for the job. It became necessary to adapt the framework to this system because the provided infrastructure was not suited for using Celery.

ISABELLE/HOL Isabelle [32] is a well known higher-order proof assistant made for interactive use. Given a user created theory it runs several different prove strategies like *sledgehammer blast* etc. on the theory and thus provides proofs. For the purpose of this thesis it was useful that Isabelle includes the strong model generators *nitpick* and *refute* which were used to generate models for axioms of the ontology. Due to the way the model generation works the strong performance is limited by amount of axiom in the problem files. However as during the usage it turned out that this tool does not fully implement the SZS ontology for the TPTP infrastructure because type errors are not reported with the status *SZS status error* and the user has to manually search the rather verbose output for the desired information.

LEO-II LEO-II [33] is an automated-theorem prover for higher-order logic which uses an cooperative approach for proving. Its extensional higher-order resolution is complemented by a first-order ATP system which is fed special representations of the axioms. In the default case this is the brainiac theorem prover E [34]. The main advantage of LEO-II in this work was that on certain axiom sets its ability to infer satisfiability was much better, meanwhile on others it ran into timeouts where *nitpick* or *Satallax* swiftly generated a model.

Satallax Satallax [35] is an automated theorem prover for higher-order logic which makes extensive use of reducing the problems to SAT problems and checking their satisfiability with *Minisat* [36]. Because the calculus to derive the SAT problems is closely tied to Henkin models it has quite strong capabilities in model generation for a set of axioms and is therefore integrated into the framework.

3.3 Inconsistency detection

3.3.1 Motivation

The knowledge collected in an expressive ontology is useless by itself. In order to be useful it must be applied to problems. However it is desirable that the

amassed knowledge is correct *in application*. Assessing the correctness of the data with applying it is hard. But since an ontology already contains so much knowledge it could be possible to find mistakes by searching through the existing knowledge. In one case the issue may be rather simple: two axioms contradict each other in an obvious way such that one is the negation of the other. It is clear that if an ontology does contain such a pair of axioms this pair will form a contradiction and thus a inconsistency. Since this is very easy to detect it is generally not found in ontologies at all. Usually the problem is more difficult. Consider the following example:

```
%TYPES
thf(peter_THFTYPE_i,type,(peter_THFTYPE_i: $i)).
thf(bill_THFTYPE_i,type,(bill_THFTYPE_i: $i)).
thf(ben_THFTYPE_i,type,(ben_THFTYPE_i: $i)).
thf(bruce_THFTYPE_i,type,(bruce_THFTYPE_i: $i)).
thf(propA_THFTYPE_o,type,(propA_THFTYPE_o: $o)).
thf(propB_THFTYPE_o,type,(propB_THFTYPE_o: $o)).
thf(father_THFTYPE_IiioI,type,(father_THFTYPE_IiioI: ($i>$i>$o))).
thf(truth_THFTYPE_IoooI,type,(truth_THFTYPE_IoooI: ($o>$o>$o))).
thf(believes_THFTYPE_IiooI,type,(believes_THFTYPE_IiooI: ($i>$o>$o))).
thf(knows_THFTYPE_IiooI,type,(knows_THFTYPE_IiooI: ($i>$o>$o))).

%AXIOMS
thf(ax1,axiom,(~ (truth_THFTYPE_IoooI @ ((father_THFTYPE_IiioI
@ bruce_THFTYPE_i @ ben_THFTYPE_i) & (father_THFTYPE_IiioI @
bruce_THFTYPE_i @ bill_THFTYPE_i)) @ $true))).
thf(ax3,axiom,(knows_THFTYPE_IiooI @ peter_THFTYPE_i @
(father_THFTYPE_IiioI @ bruce_THFTYPE_i @ ben_THFTYPE_i))).
thf(ax1126,axiom,((! [FORMULA: $o,AGENT: $i]: ((knows_THFTYPE_IiooI @
AGENT @ FORMULA) => (believes_THFTYPE_IiooI @ AGENT @ FORMULA))))).
thf(ax3303,axiom,((! [FORMULA: $o,AGENT: $i]: ((knows_THFTYPE_IiooI @
AGENT @ FORMULA) => (truth_THFTYPE_IoooI @ FORMULA @ $true))))).
```

All of the used systems return the result that this set of axioms is satisfiable even when adding a conjecture to proof to the problem set:

```
thf(con,conjecture,(believes_THFTYPE_IiooI @ peter_THFTYPE_i @
(~ @ (father_THFTYPE_IiioI @ bruce_THFTYPE_i @ bill_THFTYPE_i)))).
```

But when this conjecture is added as an axiom to the problem set with

```
thf(ax,axiom,(~ (believes_THFTYPE_IiooI @ peter_THFTYPE_i @
(~ @ (father_THFTYPE_IiioI @ bruce_THFTYPE_i @ bill_THFTYPE_i))))).
```

All automated theorem provers regard it as unsatisfiable which is counter-intuitive since in general when using production systems in artificial intelligence one can safely add the derived results to the starting problem and not turn the original axiom set unsatisfiable or wrong. One step to find such problems is to check the satisfiability of all types and axioms in the database to exclude incorrectness. Due to the large amount of axioms it is not feasible to use human heuristic

reasoning to specify conjectures for any subset. Instead one can select certain parts of the ontology and check whether they are satisfiable. This approach is highly mechanizeable and thus requires only human interaction for determining the concrete flaws in a unsatisfiable subset.

3.3.2 Theoretical considerations

Viewing an expressive ontology in **STT** as a graph is very useful for presentation and formulation of the algorithms on it. As it will be argued later the vertices can even be changed by newly acquired data and the graph can even be *inverted*. A graph $\mathbf{G}(\mathbf{V}, \mathbf{E})$ is a structure where

- $\mathbf{V}$ is a set of vertices;
- $\mathbf{E}$ is a set of edges denoted where an edge is denoted as an ordered pair.

A graph for an ontology can be defined by identifying every single type of the ontology with a vertex. Meanwhile the edges can be built by considering a relationship between the types. Most easily one can assume that two types are connected if there exists an axiom in which both of them are used. This definition of the founding relation makes the graph undirected since the relation is symmetrical. By choosing a different founding relation the graph could also be turned into a directed one. As described above the graph can also be inverted by considering the axioms as vertices and the relation two axioms are connected if there exists a type which is used in both. With this duality it is possible to choose either types or axioms as a starting point for the inconsistency detection. Also it is possible to define other relations for the ontology. Note that the graph may be restricted to only types or only axioms without using both. For instance two types or axioms could be connected when they share the same signature in **STT** or if they use the same amount of free variables. This allows to define different views on the ontology and view it from a certain aspect which can be tremendously helpful when devising new strategies for inconsistency detection or refactoring the ontology. With the above discussion one can view an expressive ontology as a structure $\mathbf{O}(\mathbf{T}, \mathbf{A}, \mathbf{R})$ where

- $\mathbf{T}$ is the set of types;
- $\mathbf{A}$ is the set of axioms;
- $\mathbf{R}$ is a set of relations between elements of $\mathbf{T}$ and $\mathbf{A}$.

Given such a structure it is possible to choose a relation $r \in \mathbf{R}$ and compute a corresponding graph. In this graph a subgraph[20] must be selected so that if a THF0 file is generated from this subgraph there exists no model[21] which

[20] A subgraph $\mathbf{S}(\mathbf{V}', \mathbf{E}')$ of $\mathbf{G}(\mathbf{V}, \mathbf{E})$ is a graph with the restriction that $\mathbf{V}' \subset \mathbf{V}$ and $\mathbf{E}' \subset \mathbf{E}$.

[21] Or there exists a countermodel.

renders the selected subgraph unsatisfiable. However the selection process can take *other*[22] graphs into consideration if the information given in the initial graph is not sufficient.
The process for turning a subgraph into a valid problem file for the automated theorem provers is also greatly simplified with this abstraction. For the generation the algorithm just needs to iterate over all the axioms used in the subgraph and collect their types. Then the types must be filtered, since some types may occur as free variables. Once the bound variables are determined, the problem file can be generated.
Furthermore this view fits nicely into the general considerations of the framework. Especially the reporting capabilities. If sensible relations are chosen the meta data divides the graph into strongly connected components which allow simple selection of the desired information. For instance the core of an upper ontology should contain the most general axioms over all possible knowledge domains. In other words the core should contain the types and axioms of the upper ontology whose interconnectedness[23] is much larger then the rest of the ontology. Of course the core can only contain a fixed number of types and axioms and therefore one must select the axioms carefully. Let r be the relation that two axioms are connected if they share a type. After having computed the graph one can just select the most *suitable* [24] axioms.
How a graph for a relation can be constructed depends on the relation but all relations must use some sort of meta data of types and axioms in order to be defined and calculated. In general this data is not present but can be derived when the ontology is read into the database or computed with all the types and axioms present.
Given the size of an expressive ontology one may naively try to check all possible subsets of the set of axioms but this approach is not feasible at all since the amount of subsets is too large and model generation being a semi-decidable function. From an abstract point of view it is clear that inconsistencies will occur more likely the more a type is constrained by the axiom it was used in. These constrains are expressed by other axioms making statements about the types appearing in the original axiom. Therefore in order to check an axiom it is not necessary to consider all possible subsets of the ontology, but only those axioms who share some connection to the axiom. This connection can be expressed by the above discussed relation between axioms and types. With this relation the search space for inconsistencies is drastically reduced since instead considering all possible subsets now only for each axiom a - in general much smaller -

[22] Therefore graphs constructed around another relation from **R**.

[23] Therefore the amount of outgoing vertices from a node.

[24] It was not taken into considerations that certain knowledge domains could contain more axioms than others and thus create more connections in the knowledge domain.. Suitable thus must mean that the axiom is used over several different knowledge domains. But for this purpose another relation must be defined. As of now the **STT** translation does not contain the necessary meta information for this relation.

amount of subsets must be considered. This amount could still be reduced since certain axioms are syntactically complete identical and contain for instance only geographical information about airports.

3.4 Practical considerations & issues

With the theoretical considerations the insertion of the ontology into the framework becomes obvious: For each type or axiom extract the local meta information like declared types, signature and so on and then box it into a dictionary with the axiom string such that the information is available under the corresponding key. Only then can the information be written into the database. Global meta information must be calculated after all axioms and types are written into the database. Since the data is contained as a document in the database, the graphs are represented implicitly over the extracted meta data. Furthermore the graphs are represented as an implicit adjacency list concerning an relation over meta data in the database. Therefore they can be constructed by querying the database. It is also very difficult to derive a general graphical user interface for the framework since the data is dynamic and it often is necessary to use scripting of some sort to retrieve information out of the ontology. Also the graphical user interface cannot be run on an remote machine which is an inconvenience when using a cluster for computation. For all of this reasons the framework uses the Python interpreter as the main interface for the user. This has the advantage that data can be manipulated at will and simple algorithms can be derived rather fast. Commonly used operations can just be added to the source file for further reuse. The standardization of prover output due to the SZS ontology [37], relives the framework from the implementation of prover specific code for parsing the results. Albeit the SZS ontology offers quite a differentiated amount of possible results, the framework currently only implements a tiny bit of it. The reasoning for it is quite simple: the framework is interested just in the satisfiability of the subsets. All other output concerning an subset is currently regarded as erroneous because there should not be any other output but *%SZS status satisfiable.* Therefore further discussion of the ontology will be omitted.
In general the workflow of the framework can be described as following: Given an expressive ontology in the THF0 file format it is inserted into the database to be used as the data source for the inconsistency detection. On this data source then the relations can be computed in order to get the graphs and subgraphs. Such a graph serves as an *discretisation* of the ontology and thus is used as the foundation for the search algorithm although any search algorithm can be used for this purpose. The search algorithm is currently rather simple and just searches the whole space and generates for every possible combination of axioms and relevant types a problem file. More sophisticated algorithms are possible but the memory overhead does not seem to justify the possible speed up, since the worst case remains: there are no inconsistencies. These problem files are then used as an input for the automated theorem provers. Using the SZS ontology of the

TPTP platform the results are parsed and if the set is not satisfiable then it will be inserted into a special document in the database. This workflow is seen in Figure 1.

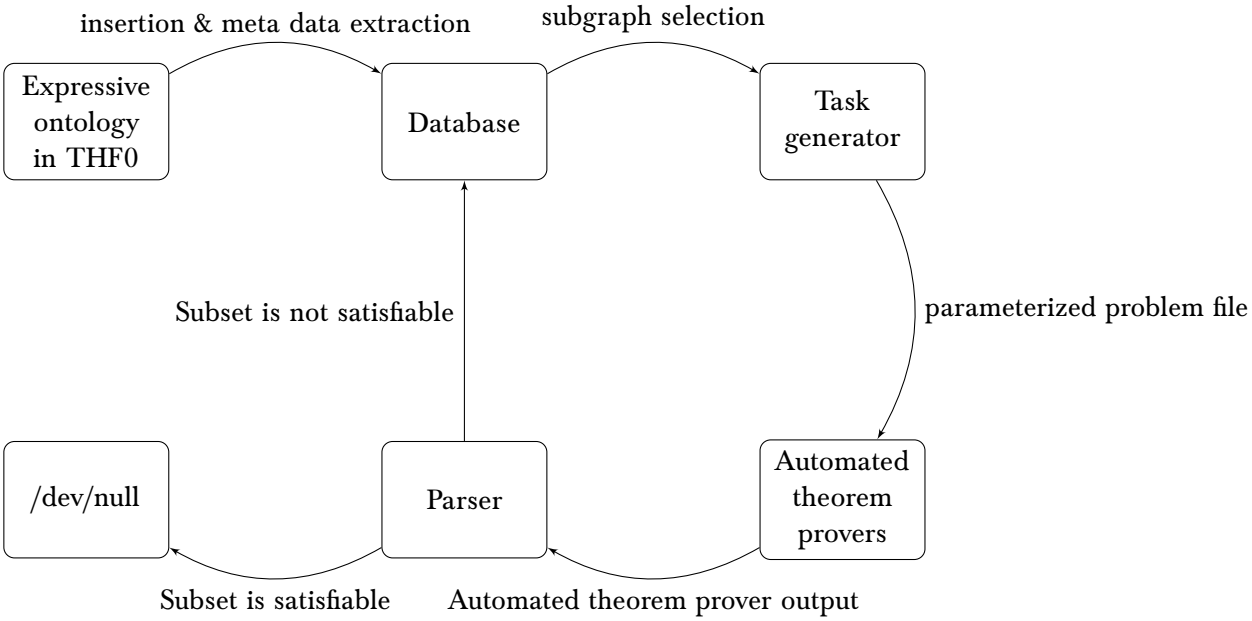

Figure 1: Workflow of the framework

With the figure of the framework its resemble to a unix pipeline is striking. It can be seen as a system of pipes and individual commands mapping an ontology to a set of finite models where as only the unsatisfiable axiom sets are recorded.
When considering subsets checking one is presented with a conflict of interests: Even though one is interested in finding an inconsistency one prefers *simple* inconsistencies - therefore inconsistencies which have the minimal amount of axioms and types - over inconsistencies which have a lot of axioms and types. However latter inconsistencies are easier to find since the constraints for individual types are more strict. In the framework this issue is addressed by allowing to start either from the top or from the bottom for the inconsistency detection. However the choice is arbitrary for small data sets but can be helpful on large data sets.
Generating all possible subsets concerning a certain relation for an axiom in memory requires a lot of memory if the relation is large. In order to solve this issue and still allow simple creation of new algorithms a somewhat neglected feature of Python had to be used: Generators. Generators in Python are a combination of the lazy list of Haskell with the well known iterator pattern. Like lazy lists the next value of generator will be only calculated when needed. But unlike lazy lists, generators do not keep previous values in memory. After a value was used it will be discarded. The later feature resembles iterators with no knowledge about their previous state. With the generators it is very convenient

to express the generation of large data sources and their consumption and limit memory consumption at the same time. However because no state is known at the time of the execution manual state keeping has to be done in order to assess the progress of a checking process. Subset generation is done step wise in the framework. In order to generate all of the power set for a set of axioms, the power set for set with equal size to cardinality of the set of axioms and numbers as symbols for the axioms is expressed as generator comprehension and then by using a dictionary - which constantly maps each occurring symbol in the power set to an axiom - mapped into the axioms space. Furthermore if a checking process is terminated abnormally due to some error and the last generated subset is not known then the whole checking process must be started again. However if it is known, then one can simply skip the previous n subsets and continue.
Albeit Python is usually fast enough for daily applications there are bottlenecks when dealing with loops. With the standard implementation of the Python interpreter it evaluates the loop header and loop body separately. Therefore optimization to the loop can not be done. In order to allow optimization of the loop the loop must be refactored to a generator comprehension. Once more these are very similar to the Haskell list comprehension syntax. The first version of the framework was implemented using loops and was very slow for subset generation and used a lot of memory. After refactoring the program several times - mainly successive replacement of loops with list comprehensions and then generator comprehensions - the runtime could be reduced to $\frac{1}{3}$ of the previous runtime with *practical* [25] constant memory consumption. Note that the usage of for-looping only is problematic when dealing with computation intensive tasks. The speed is fine when dealing with I/O based tasks.
Another optimization had to be made concerning the database. Since data had to be present for making problem files for the provers a early version used to create a connection for the database for every single data access and close it directly afterwards. The overhead of the BSON-translation of the data and the constant connection and disconnection decelerated the whole framework even though memory consumption was kept to a minimum. By introducing an optional parameter to reuse an existent connection this issue was only partially remedied. The big speed up was achieved by trading memory for computation time as well as latency and thus keeping a cache of the database entries for axioms and types locally so that they could be retrieved directly.
Of course it is pointless to save all the data generated by the automated theorem provers since with the current subset method the database will run out of memory quite fast. Additionally not all the output is interesting at all. Most of internal output of the automated theorem prover is not standardized or uniform and thus cannot be used to derive useful information for the further proving process. However one is interested in the whole output when dealing with an unexpected return value. Therefore in this case the whole output of the automated theorem

[25] It is bounded by the size of the largest subset and therefore the total size of the subgraph.

prover is written into the database for further investigation.
The lack of the standardization continues when confronted with the return codes of the automated theorem provers. Each of them has a custom set of return code in order to signal the success or failure when proving. With the framework these are taken into consideration by using the *sh* module from the Python package repository. However this issue only matters when running the framework on the local machine since in general cluster applications only care about things written to the standard streams.
The employed automated theorem provers are put into subdirectories of the directory where the framework resides. Their configuration is managed by the *sh* module and abstracted in their own modules.
The reporting capabilities are added by using the library matplotlib [38] from the Python package repository.

3.5 Modes

As of now the frameworks supports three different kinds of running modes: a local mode, a remote mode with Celery and a remote mode with TORQUE. Even the last two modes allow to use a cluster for computation the way interprocess communication is realized differs. Even though the concrete stamping of the architecture depends on the running mode it just shifts the different modules around. In general the architecture looks like in Figure 2.

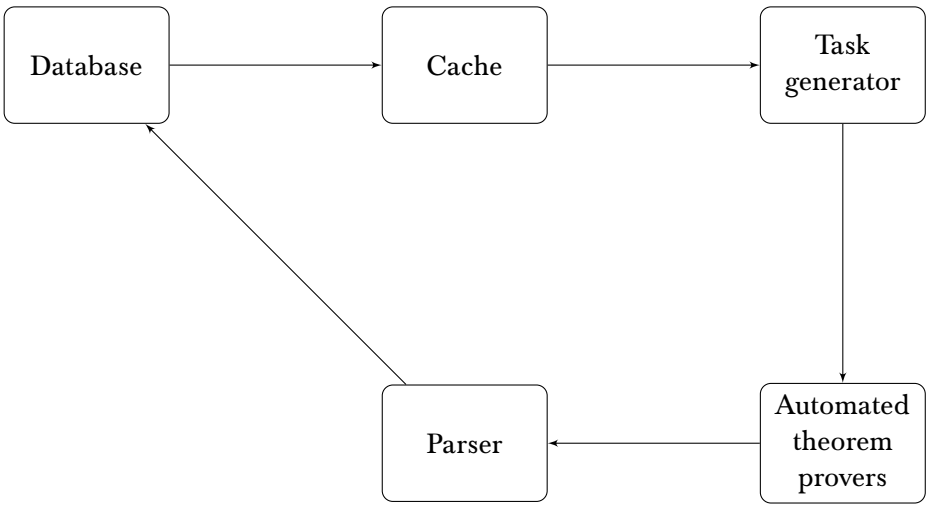

Figure 2: General architecture

As the name implies the local mode reduces all the operations onto the local machine, which servers as database and task generator as well as computation platform for the different provers. This mode allows to easily test different algorithms and is used to derive the configuration for the specific provers. It is also possible to specify another host for the database, so that the data storage is

on another machine and different hosts can use it to cooperatively analyze the ontology.
The remote mode with Celery allows to spread the computations onto a cluster of more potent machines. Although in general the local machine still serves as the task generator, the tasks are cached on the RabbitMQ-server - which may run on another machine - and dispatched from there. Also the output of the tasks is parsed on the nodes and then written into the database because the nodes share all the information of the task generator. The remote mode with Celery is not tested as much as the mode with TORQUE since development had to be discontinued when it become clear that the assigned infrastructure of the computer science department could not support it.
The remote mode with TORQUE achieves a similar setup as the remote mode with Celery. However where Celery simplifies the deployment of tasks by implicit data sharing, TORQUE does not offer similar ways or a Python interface. Therefore a simple queue for task creation and parsing the results had to be developed. The queue uses two states for classifying states: a list for the job still to be computed and a list for the jobs which are done. Since the TORQUE system does not signal whether the job finished the file system is used for interprocess communication in this case. Since TORQUE uses shellscripts to distribute the calculations on the cluster a simple templating engine was used to instantiate a calculation job. Each of the jobs is in its own directory with all the necessary data for the job. After finishing the job a file with the name *DONE* is created to mark the job as done. With this tag on the next iteration of the task scheduler the result file will be parsed and inserted into the database if an unexpected result was detected. Due to the way the cluster is implemented special attention must be given to the job size. When using the intuitive division of the subsets into jobs - therefore each subset forms a job - the cluster starts to crumble under the load. Depending on the size of the job the overhead for initiating a job is greater than the cost of computing it. This constitutes a severe limitation for the total throughput of the system, since the system is busy doing I/O-operations instead of computing. Furthermore this causes a very high load on the underlying network file system which in this mode is used for interprocess communication between the program and the computation jobs.
The constant creation of locks and their releases puts a very high load on the network. All of these issues make it necessary to build small packages of jobs and let the system compute these chunks while limiting the I/O-operations. Since *nitpick* is the strongest but slowest model generator - on empiric basis it generates most reliably models for small axiom sets - a two step approach was implemented: First the axioms are checked with *nitpick* and only if *nitpick* cannot deduce a result for the axiom set because it runs into a timeout, then other tools will be used. With this approach it is also necessary to introduce a more sophisticated task queue - described in Figure 3 - since now there are two kind of jobs: single jobs and compound jobs. Usually the compound jobs consist of many smaller jobs for *nitpick* while the other single job consists of a call to an automatic theorem

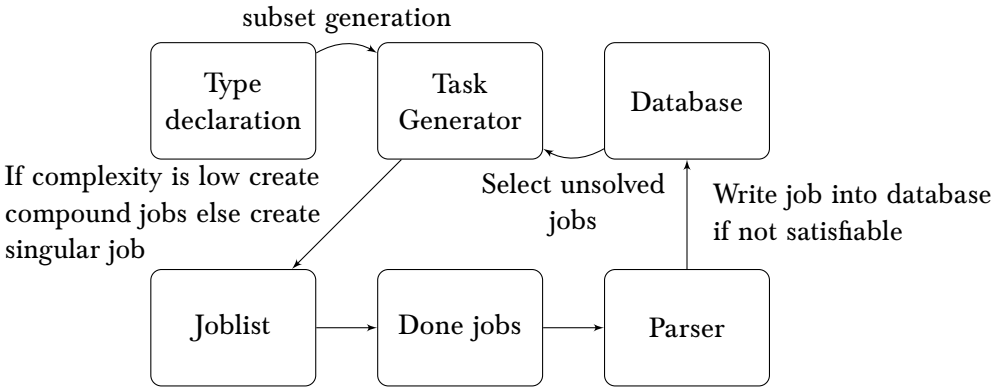

Figure 3: The framework's task scheduler for TORQUE

prover which is not restricted to axiom sets of small size. For determining the single tasks the results from the compound job are written into a special document in the database with a tag and polled by the single job generating task.
Another issue comes up when regarding the output of the automated theorem prover: albeit all of the used systems support to use a list of files as input their output for this list is *en block*. This makes it impossible to determine the individual status of each subset. Just filtering out the SZS ontology results still leaves a large search space because one can only derive that at least one of the problem files does contain an undesirable result. Therefore the job file for the cluster contains a separate call of the prover for each problem file divided by a special sequence of symbols to recognize and differ between the output for the problem files. However this approach has drawbacks. The computation cost for initializing the theorem prover is paid for each and every single call. In case of *nitpick* - which first of all starts a Java Virtual Machine and launches ML code on top of it - this cost is substantial.

3.6 Categorization of detected inconsistencies

3.6.1 Broken types

One of the runs started with generating a model for each axiom in the complete SUMO. During this run a pattern of broken types emerged: The type *uknTP* is generated even though it should not be existent at all . For instance:

```
%Type Declarations
thf(lSymbolicString_THFTYPE_i,type,(lSymbolicString_THFTYPE_i: $i)).
thf(instance_THFTYPE_IiioI,type,(instance_THFTYPE_IiioI: ($i>$i>$o))).
%Axioms thf(ax9242,axiom,((! [TOKEN: $i,X: uknTP]:
((instance_THFTYPE_IiioI @ TOKEN @ lSymbolicString_THFTYPE_i))))).
```

It was not possible to map this axiom back into SUMO, since it is not clear why the X occurs and the next fitting formula in SUO-KIF does not contain the X

at all. Furthermore it is dubios how the X and its type are derived at all. The approach described in [4] should not produce such results. It is safe to assume that either the translation process from SIGMA is flawed or SUMO contains serious mistakes concerning *TOKEN*. This mistake discovered a two-fold inconsistency. It showed that all of the used provers but *nitpick* do not typecheck variables which are not used in the formula. Furthermore the other provers assumed that since the type is not used it can be ignored and accepted the input and even delivered the status satisfiable for the axioms. This is clearly undesireable behaviour and contradiction the specifications. All of this errors occur when *TOKEN* or *MAP* is involved. In order to determine the source of this error it would be helpful if the translation with the SIGMA environment would include the original formula as a comment or annotation in the translation. This would allow the user to avoid the tedious search in the original ontology. For a complete list of this kind of errors consider the appendix A.1.

3.6.2 Incorrect semantic modeling

By using the type system introduced by **STT** it was possible to discover errors in the semantic modeling in SUMO. Even though once more the type *uknTP* was generated and used, the matter is different than from the broken types issue. This time it is clear that the type *should* be in the axiom and that the problem is somewhere else. Consider the example below:

```
%Type Declarations
thf(lDeadFn_THFTYPE_IiiI,type,(lDeadFn_THFTYPE_IiiI: ($i>$i))).
thf(lFish_THFTYPE_i,type,(lFish_THFTYPE_i: $i)).
thf(subclass_THFTYPE_IiioI,type,(subclass_THFTYPE_IiioI: ($i>$i>$o))).
thf(instance_THFTYPE_IiioI,type,(instance_THFTYPE_IiioI: ($i>$i>$o))).
thf(lSeafood_THFTYPE_i,type,(lSeafood_THFTYPE_i: $i)).
thf(part_THFTYPE_IiioI,type,(part_THFTYPE_IiioI: ($i>$i>$o))).
%Axioms thf(ax6670,axiom,((! [S: $i,F: $i,M: $i]:
((instance_THFTYPE_IiioI @ S @ lSeafood_THFTYPE_i) => (? [DA: $i,AC:
uknTP]: (instance_THFTYPE_IiioI @ DA @ (lDeadFn_THFTYPE_IiiI @ F)) &
part_THFTYPE_IiioI @ M @ DA) & (subclass_THFTYPE_IiioI @ F @
lFish_THFTYPE_i)))))).
```

It is possible to map the formula to a SUO-KIF representation:

```
(=> (instance ?S Seafood) (exists (?DA ?AC) (and (instance ?DA (DeadFn
    ?F)) (part ?M ?DA) (subclass ?F Fish))))
```

From the documentation of the formula it is visible that *AC* should be used in the second part but is not. One can safely assume that this error stems from an oversight when formulating this axiom. Probably one of the *DA*s should

be an *AC*. It is interesting that these kinds of semantic errors are exposed by the inconsistency detection process by just validating the types and finding inconsistencies and errors. Of course the previous discussion concerning the behavior of provers applies in this case too. For a complete list this kind of errors consider the appendix A.2.

4 Further work

Due to the way the model generation process works, even small changes in the founding axiom set can lead to big changes in the resulting model and thus prevent to reuse the information gained from earlier computations. Given the current state of model theory it could help immensely to find relationships between models so that at least a part of - or better the whole previous computation - can be used to speed up the whole process.
Even though the developed framework gets the job done, it is still rough. Further work includes polishing of the framework and applying it to other expressive ontologies, so that new common patterns of usage are discovered as well as new relations and be added to the framework. With further detected inconsistencies it could be possible to specify the conditions in which inconsistencies occur and use a more guided search by formulating heuristics.
As of writing this the checking process for several subsets is not done. Therefore it is expected that until the presentation of my thesis some new results will be found.

5 Conclusion

The implemented framework for inconsistency detection implements a viable approach for inconsistency detection in expressive ontologies. Due to lacking computation time the found inconsistencies are not that exciting. Nevertheless it was possible to spot errors in SUMO and probably even the translation algorithm to **STT** from the SUO-KIF representation.

A List of found errors

A.1 Broken Types

This kind of error is usually rather small. Therefore just a listing of the found errors will be given and the *bad* type shortly mentioned.

- `thf(ax5420,axiom,((! [AGENT: $i,TYPE: uknTP]: (((instance_THFTYPE_IiioI @ AGENT @ lOrganization_THFTYPE_i) | (instance_THFTYPE_IiioI @ AGENT @ lGeopoliticalArea_THFTYPE_i))))).`

  TYPE is declared but never used.

- `thf(ax9242,axiom,((! [TOKEN: $i,X: uknTP]: ((instance_THFTYPE_IiioI @ TOKEN @ lSymbolicString_THFTYPE_i))))).`

  X is declared but does not find application in the axiom at all.

- `thf(ax18776,axiom,((! [TOKEN: $i,X: $i,MAP: uknTP]: ((represents_THFTYPE_IiioI @ TOKEN @ X))))).`

  MAP is declared but not used further. There is however a similar formula in SUMO:

  ```
  (=>
      (codeMapping ?MAP ?TOKEN ?X)
      (represents ?TOKEN ?X))
  ```

  However this uses code mapping as a wrapper around the types in the axiom. Nevertheless it could help for further error search.

A.2 Semantic modeling errors

This section follows the following pattern: the erroneous axiom followed by the SUO-KIF representation and a short description of what went wrong.

- **THF0** `thf(ax1349,axiom,((! [ROOM: $i,TI: $i,CUST: $i,AGENT: $i,HOTEL: $i,HP: $i,NUM: $i]: (((propositionOwner_THFTYPE_IiioI @ HP @ AGENT) & (instance_THFTYPE_IiioI @ HP @ lHotelPackage_THFTYPE_i) & (accommodationProvider_THFTYPE_IiioI @ HP @ HOTEL) & (roomStay_THFTYPE_IiiioI @ HP @ NUM @ ROOM) & (buys_THFTYPE_IiiioI @ CUST @ AGENT @ HP)) => (? [T: uknTP]: ((TI = (lMeasureFn_THFTYPE_IiiiI @ NUM @ lDayDuration_THFTYPE_i)) & (holdsDuring_THFTYPE_IiooI @ TI @ (guest_THFTYPE_IiioI @ CUST @ HOTEL)))))))).`

  SUO-KIF The error can be mapped to Hotel.kif 2670-2681:

  ```
  (=>
      (and
  ```

```
        (propositionOwner ?HP ?AGENT)
        (instance ?HP HotelPackage)
        (accommodationProvider ?HP ?HOTEL)
        (roomStay ?HP ?NUM ?ROOM)
        (buys ?CUST ?AGENT ?HP))
    (exists (?T)
        (and
            (equal ?TI
                (MeasureFn ?NUM DayDuration))
            (holdsDuring ?TI
                (guest ?CUST ?HOTEL)))))
```

Description The type *T* is introduced but not used further. It can be assumed that either the *TI* is the *T* or one of them should be replaced.

- **THF0** `thf(ax6670,axiom,((! [S: $i,F: $i,M: $i]: ((instance_THFTYPE_IiioI @ S @ lSeafood_THFTYPE_i) => (? [DA: $i,AC: uknTP]: ((instance_THFTYPE_IiioI @ DA @ (lDeadFn_THFTYPE_IiiI @ F)) & (part_THFTYPE_IiioI @ M @ DA) & (subclass_THFTYPE_IiioI @ F @ lFish_THFTYPE_i)))))).`

  SUO-Kif It can be mapped to SUO-KIF in Food.kif 845-851

```
(=>
    (instance ?S Seafood)
    (exists (?DA ?AC)
        (and
            (instance ?DA
                (DeadFn ?F))
            (part ?M ?DA)
            (subclass ?F Fish))))
```

  Description *AC* is not referenced after the declaration. According to the documentation it could be that it is part of *DA* instead of *M*.

- **THF0** `thf(ax19111,axiom,((! [M1: $i,M2: $i,X: $i,NUM2: $i,NUM1: $i]: ((instance_THFTYPE_IiioI @ X @ lCurrencyExchangeService_THFTYPE_i) => (? [AMT: $i,AMT1: uknTP,CURR1: $i,AMT2: uknTP,CURR2: $i,CUST: $i]: ((agent_THFTYPE_IiioI @ X @ CUST) & (~ @ (CURR1 = CURR2)) & (instance_THFTYPE_IiioI @ AMT @ lCurrencyMeasure_THFTYPE_i) & (holdsDuring_THFTYPE_IiooI @ (lWhenFn_THFTYPE_IiiI @ (lStartFn_THFTYPE_IiiI @ X)) @ ((monetaryValue_THFTYPE_IiioI @ M1 @ AMT) & (possesses_THFTYPE_IiioI @ CUST @ M1) & (AMT = (lMeasureFn_THFTYPE_IiiiI @ NUM1 @ CURR1)))) & (holdsDuring_THFTYPE_IiooI @ (lWhenFn_THFTYPE_IiiI @ (lEndFn_THFTYPE_IiiI @ X)) @ ((monetaryValue_THFTYPE_IiioI @ M2 @`

```
AMT) & (possesses_THFTYPE_IiioI @ CUST @ M2) & (~ @
(possesses_THFTYPE_IiioI @ CUST @ M1)) & (AMT =
(lMeasureFn_THFTYPE_IiiiI @ NUM2 @ CURR2))))))))).
```

SUO-KIF It can be mapped to Hotel.kif 2030-2049.

```
(=>
    (instance ?X CurrencyExchangeService)
    (exists (?AMT ?AMT1 ?CURR1 ?AMT2 ?CURR2 ?CUST)
        (and
            (agent ?X ?CUST)
            (not
                (equal ?CURR1 ?CURR2))
            (instance ?AMT CurrencyMeasure)
            (holdsDuring
                (WhenFn
                    (StartFn ?X))
                (and
                    (monetaryValue ?M1 ?AMT)
                    (possesses ?CUST ?M1)
                    (equal ?AMT
                        (MeasureFn ?NUM1 ?CURR1))))
            (holdsDuring
                (WhenFn
                    (EndFn ?X))
                (and
                    (monetaryValue ?M2 ?AMT)
                    (possesses ?CUST ?M2)
                    (not
                        (possesses ?CUST ?M1))
                    (equal ?AMT
                        (MeasureFn ?NUM2 ?CURR2)))))))
```

Description AMT1 is declared but not used. Probably a renaming must be done.

- **THF0** `thf(ax11578,axiom,((! [MOTION: $i,HORSEBACK: $i]: ((instance_THFTYPE_IiioI @ HORSEBACK @ lEquitation_THFTYPE_i) => (? [HORSE: uknTP]: ((instance_THFTYPE_IiioI @ MOTION @ lHorseRiding_THFTYPE_i) & (subProcess_THFTYPE_IiioI @ MOTION @ HORSEBACK)))))).,axiom,((! [MOTION: $i,HORSEBACK: $i]: ((instance_THFTYPE_IiioI @ HORSEBACK @ lEquitation_THFTYPE_i) => (? [HORSE: uknTP]: ((instance_THFTYPE_IiioI @ MOTION @ lHorseRiding_THFTYPE_i) & (subProcess_THFTYPE_IiioI @ MOTION @ HORSEBACK)))))).`
  `thf(ax11578,axiom,((! [MOTION: $i,HORSEBACK: $i]:`

```
((instance_THFTYPE_IiioI @ HORSEBACK @ lEquitation_THFTYPE_i) => (?
[HORSE: uknTP]: ((instance_THFTYPE_IiioI @ MOTION @
lHorseRiding_THFTYPE_i) & (subProcess_THFTYPE_IiioI @ MOTION @
HORSEBACK))))))).
```

SUO-KIF
```
(=>
    (instance ?HORSEBACK Equitation)
    (exists (?HORSE)
        (and
            (instance ?MOTION HorseRiding)
            (subProcess ?MOTION ?HORSEBACK))))
```

Description Horse is declared but not used in the subsequent expression at all.

B Sourcecode of the framework

The sourcecode of the framework. Besides this listing and the included CD it is also available as a tar.gz from http://arch.cassiopeia.uberspace.de/ragozin.tgz. Note that the framework was developed under Linux and therefore probably will only run under unices. However in order to port it to Windows probably it will just be necessary to change the calls to the os module. The included CD contains a small README on installation and running the framework.

```
from celery import Celery
import sh
import os
import satallax
import nitpick
import leo2
import random
import string
import pymongo as p
import re
import itertools as i
import cProfile
import functools
import tenjin as t
import time
from tenjin.helpers import *
t.set_template_encoding('cp932')
engine = t.Engine()
import shutil

# The essential database configuration

DB = {
'mongo': {
    'DB_NAME':'SUMO',
    'DB_HOST' : 'ontology',
    'DB_PORT' : 27017
     }
}

# Set the torque job directory
TOJ_DIR = '/home/bude/mziener/jobs/'

# Global variables for caching
AXIOMS = {}
TYPES = {}
GAXIOMS = {}
TYPES = {}
GTYPES = {}
# The database driver of MongoDB cannot handle arbitrary large chunks of data
BULK_SIZE = 50000

# Compile once run often.
IS_AXIOM = re.compile("^.*,axiom,.*")
IS_THFTYPE = re.compile("^.*,type,.*")
DECLARATIONS = re.compile('^.*\((?P<name>\w+): *(?P<signature>[^0-9]+)\)\).')
USED_TYPES = re.compile('[\( @](\w+)[\) ]')
DIVIDER = re.compile('\[@DIVIDER@\]')

# The prover configurations
provers = {
    'satallax':
    {
        'call': satallax.satallax.bake('-N', _ok_code=[1,2,15]),
        'satout': satallax.SATISFIABLE

    },
    'nitpick' :
    {
        'call': nitpick.nitpick.tptp_nitpick.bake(600, _ok_code=[1,2]),
        'satout': nitpick.SATISFIABLE,
    },
    'leo2':
    {
        'call': leo2.leo2.bake( _ok_code=[0,6,7]),
        'satout': leo2.SATISFIABLE,
    }
```

```
}

# Temporary directory for local mode
TMPDIR = os.getcwd()+'/tmp/'

celery = Celery('worker',
                broker='amqp://',
                backend='amqp://')


def random_string(n):
    """ Generate a random string of length n for creating a file """
    return ''.join(random.choice(string.ascii_uppercase + string.digits) for x in range(n))

def connectToDB(db_entry):
    """ Given an entry of DB construct the connection and return the connection
    so it can be used by other functions.
    """
    c = p.MongoClient(DB[db_entry]['DB_HOST'],
                      DB[db_entry]['DB_PORT'])
    db = c[DB[db_entry]['DB_NAME']]
    return db

def cleanDB(ontologyname, db='mongo'):
    """Clean up the database so that its useable for another run with
another ontology"""
    db = connectToDB(db)
    db.drop_collection(ontologyname+"_subsets")
    db.drop_collection(ontologyname+"_axioms")
    db.drop_collection(ontologyname+"_types")
    db.drop_collection('SUMO')

def insertOntologyToDB(ontologyname, ontologyfile, db='mongo'):
    """ Given a ontology file in thf0 this function
    assumes that matches each line matches a theorem or axiom.
    It constructs database entries for future uses. """
    with open(ontologyfile, encoding='utf-8') as f:
        i = 0
        j = 0
        db = connectToDB(db)
        axioms = []
        types = []
        for line in f.readlines():
            line = line.rstrip()
            if re.search('^%.*',line) or (line == ''):
                pass
            else:
                if re.match(IS_AXIOM, line):
                    i += 1
                    axioms.append(
                        {'id':i,
                     'data':line,
                     'used_types': getUsedTypes(line)
                         })
                if re.match(IS_THFTYPE, line):
                    j += 1
                    types.append(
                        {'data':line,
                         'declared_type': getTypeDeclaration(line),
                         'id': j
                         })
        # On large ontolgoies the insert must be done with BULKSIZE
        db[ontologyname+'_axioms'].insert(axioms)
        db[ontologyname+'_types'].insert(types)
        return True, i,j
    return False

def getTypeDeclaration(data):
    """ Extract the type information out of type declarations.
    Returns a dictionary with name and signature field. """
    r = re.search(DECLARATIONS, data)
    return {'name': r.group(1),'signature':r.group(2)}

def getUsedTypes(data):
    """ Extract the used types out of a axiom. If it is parametrized the
    parameters are extracted too. """
    return USED_TYPES.findall(data)


def createSubsets(type_declaration, db_entry, db_con=None, reverse=False):
    """ This functions creates the subsets for a type_declaration."""
    d = reenumerateAxioms(AXIOMS.values())
    subsets = (mapIntoAxioms(d, s,  db_entry, db_con) for s in allsubsets(len(AXIOMS.keys()),
          reverse) if s!=[])
```

```
    return subsets

def mapIntoAxioms(mapping, subset, db_entry, db_con=None):
    """ Given a mapping return the relevant axioms corresponding to the mapping
    """
    return (AXIOMS[mapping[i]]['id'] for i in subset)

def getAxiomsWithType(type_name, db_entry, db_con=None):
    """ Searches the database for a type_name and returs all matches
    """
    db = db_con if db_con else connectToDB(db_entry)
    return (db[DB[db_entry]['DB_NAME']+'_axioms'].find({'used_types': type_name}))

def reenumerateAxioms(axiom_list):
    """ For subsetgeneration we must first reenumerate the axioms in order to
    be able to use small integers as symbols for the generations process and then map
    them to the axioms. This function return a dictionary with the exact mapping between
    the integer and the id of the axiom. NOTE: that is assumed that the axioms ids are unique.
    """
    return {k+1:v['id'] for k,v in enumerate(axiom_list)}

def allsubsets(n, reverse=False):
    """ Given a number returns all subsets containing combinations of 1..n"""
    if reverse:
        return i.chain(*(i.combinations(range(1,n+1), ni) for ni in reversed(range(1,n+1))))
    return i.chain(*(i.combinations(range(1,n+1), ni) for ni in range(1,n+1)))

def createGlobalCache(db_entry, db_con=None):
    """ Creates the global cache. Note that this function should be called before doing
          anything else """
    db = db_con if db_con else connectToDB(db_entry)
    global GAXIOMS
    GAXIOMS = {a['id']:a for a in db[DB[db_entry]['DB_NAME']+'_axioms'].find()}
    global GTYPES
    GTYPES = {t['declared_type']['name']:t for t in db[DB[db_entry]['DB_NAME']+'_types'].find()
          }

def createLocalCache(type_declaration):
    """ A smaller version of the cache for mapping the subsets into the axiom space """
    global AXIOMS
    AXIOMS = {a['id']:a for a in GAXIOMS.values() if type_declaration['declared_type']['name']
          in a['used_types']}

def generatePersistentProblemsForType(type_declaration, db_entry, db_con=None):
    """ This functions binds everything together so that it will be possible to
    generate all the for a type and thus to generate all the problems just to iterate over the
    types. Note that this function writes all the problems into the database.
    """
    db = db_con if db_con else connectToDB(db_entry)
    j = 0
    if 1<=len(AXIOMS.keys())<=16:
        db[DB[db_entry]['DB_NAME']+'_subsets'].insert(
            ({'axiom_ids': list(s)} for s in createSubsets(type_declaration, db_entry, db)))
    elif len(AXIOMS.keys())>0:
        sets = ({'axiom_ids': list(s)} for s in createSubsets(type_declaration, db_entry, db))
        limit = 2**len(AXIOMS)
        while j*BULK_SIZE<limit:
            print('INSERTED', (j*BULK_SIZE)/limit)
            j+=1
            db[DB[db_entry]['DB_NAME']+'_subsets'].insert(i.islice(sets, BULK_SIZE), timeout=
                False)

def generateProblemsForType(type_declaration, db_entry, db_con=None, reverse=False):
    db = db_con if db_con else connectToDB(db_entry)
    createGlobalCache('mongo')
    createLocalCache(type_declaration)
    return ({'axiom_ids': list(s)} for s in createSubsets(type_declaration, db_entry, db,
          reverse))

def expandSubsetWithProoverInformation(axiom_ids, prover_name, result, output, db_entry, db_con
    =None):
    """ Given a list of axiom_ids and a prover_name, add entries prover_name+_result,
          prover_name+output. """
    db = db_con if db_con else connectToDB(db_entry)
    db[DB[db_entry]['DB_NAME']+'_subsets'].update({'axiom_ids': axiom_ids},
                                                  {'$set':
                                                   {
                                                    prover_name+'_result': result ,
                                                    prover_name+'_output': output}
                                                })
    return True

def getSubsetByAxiomIds(axiom_ids, db_entry, db_con=None):
    db = db_con if db_con else connectToDB(db_entry)
```

```
    return db[DB[db_entry]['DB_NAME']+'_subsets'].find_one({'axiom_ids': axiom_ids})

def getTypeImportance(type_declaration, db_entry, db_con=None):
    """ The importance of a type is its interconnectivity """
    db = db_con if db_con else connectToDB(db_entry)
    return len(list(db['SUMO_axioms'].find(
        {'used_types':type_declaration['declared_type']['name']})))

def checkAllTypes(db_entry, use_treshold=False):
    db = connectToDB(db_entry)
    types = db[DB[db_entry]['DB_NAME']+'_types'].find(timeout=False)
    for td in types:
        createLocalCache(td)
        print(len(AXIOMS))
        if use_treshold:
            if getTypeImportance(td, db_entry, use_treshold, db):
                generateProblemsForType(td, db_entry, db)
        else:
            generateProblemsForType(td, db_entry, db)

def getTypes(axiom_ids):
    """ Given axiom ids return tall the used types without duplicates """
    return list(set([GTYPES[type_name]['data'] for axiom_id in axiom_ids['axiom_ids']
            for type_name in GAXIOMS[axiom_id]['used_types']
                    if type_name in GTYPES]))


@celery.task
def runPersistentProve(prover, axiom_ids, db_entry):
    """ Like all the functions with Persistent in it this writes all the stuff to the database.
    It can be useful for testing purposes"""
    db = connectToDB('mongo')
    db_en = db['SUMO_pers'].find_one({'axiom_ids': axiom_ids})
    print(axiom_ids)
    if db_en!=None and (prover+'_result' in db_en):
        pass
    else:
        problem = createTHFStringFromProblemSet(getTypes(axiom_ids), axiom_ids)
        filename = writeProblemFile(problem)
        out = provers[prover]['call'](filename)
        out = str(out)
        prover_result = True if len(provers[prover]['satout'].findall(out))>0 else False
        os.unlink(filename)
        db['SUMO_pers'].insert({'axiom_ids': axiom_ids,
                                'problem': problem,
                                prover+'_output': out,
                                prover+'_result': prover_result})
    return True

def runPersistentProveOnType(type_declaration, prover, db_entry, reverse=False, remote=True):
    """ Function to leverage runPersistentProve into looping over all types """
    db = connectToDB('mongo')
    createGlobalCache('mongo')
    for axiom_ids in generateProblemsForType(type_declaration, db_entry, db, reverse):
        if remote:
            runPersistentProve.apply_async(prover, axiom_ids, db_entry)
        else:
            runPersistentProve(prover, axiom_ids, db_entry)

def writeProblemFile(contents):
    """ Write a problem file for local mode """
    filename = TMPDIR+random_string(10)
    while os.path.exists(filename):
        filename = TMPDIR+random_string(10)
    problemfile = open(filename, 'w+')
    problemfile.write(contents)
    return problemfile.name

def writeProblemFileForTorque(contents):
    """ Given the contens to write this function generates an entry in  the
    file system as a reference point for the torque cluster. """
    jobname = random_string(25)
    while os.path.exists(TQJ_DIR+jobname):
        jobname = random_string(25)
    jobdir = '/'.join([TQJ_DIR,jobname])
    os.makedirs(jobdir)
    problemfile = open('/'.join([jobdir, jobname]), 'w+')
    problemfile.write(contents)
    return jobname


def writeMultiProblemFileForTorque(mjobname, problems):
    """ Handle the creation of multiple jobs """
    jobdir = '/'.join([TQJ_DIR, mjobname])
```

```
    g = filenameGenerator('problem','')
    names = []
    os.makedirs(jobdir)
    for problem in problems:
        name = next(g)
        problemfile = open('/'.join([jobdir, name]), 'w+')
        problemfile.write(problem)
        names.append(name)
    return names

def writeJobFileForTorque(contents,jobname):
    """ The single version of previous function """
    jobdir = '/'.join([TQJ_DIR, jobname])
    jobfile = open('/'.join([jobdir, 'job.sh']),'w+')
    jobfile.write(contents)

def removeProblemFromTorque(jobname):
    """ Clean up after jobname was done. """
    path = '/'.join([TQJ_DIR,jobname])
    shutil.rmtree(path)


def prepareProblemSet(type_declarations, axiom_ids):
    """ Given a list of tuples of the form (type_declaration, axiom) generate the data for
    the higher-order logic system"""
    type_ids = (t for type_ids in thfaxioms for sl in type_ids[0]  for t in sl)
    axiom_ids = (t[1]['data'] for t in thfaxioms)
    type_ids = list(set(type_ids))
    axiom_ids = list(set(axiom_ids))
    return {'axiom_ids': axiom_ids, 'type_ids': type_ids}



def createTHFStringFromProblemSet(type_declarations, axiom_ids):
    """ Given a clean problem set, create the string for the higher order system and write
    the entry in the database for the subset. """

    string = '%Type Declarations\n'
    string += '\n'.join(type_declarations)
    string += '\n%Axioms \n'
    string += '\n'.join([GAXIOMS[axiom_id]['data'] for axiom_id in axiom_ids['axiom_ids']])
    return string


@celery.task
def runBlitzProve(prover, axiom_ids, db_entry):
    """ Same as persistent proving but throw all the results away if they are needed """
    db = connectToDB(db_entry)
    db_en = db['SUMO_blitz'].find_one({'axiom_ids': axiom_ids})
    if db_en!=None and (prover+'_result' in db_en):
        pass
    else:
        problem = createTHFStringFromProblemSet(getTypes(axiom_ids), axiom_ids)
        filename = writeProblemFile(problem)
        out = provers[prover]['call'](filename)
        out = str(out)
        prover_result = True if len(provers[prover]['satout'].findall(out))>0 else False
        if 'post' in provers[prover] and False:
            try:
                out = provers[prover]['post'](out)
                print(out)
            except:
                pass
        os.unlink(filename)
        if prover_result:
            pass
        else:
            db['SUMO_blitz'].insert({'axiom_ids': axiom_ids,
                                     'problem': problem,
                                     prover+'_output': out,
                                     prover+'_result': prover_result})


def runBlitzProveOnType(type_declaration, prover, db_entry, reverse=False, remote=False):
    """ Leverage previous function for looping with some additional configuration options.
    With remote=True this functions uses Celery for computations """
    db = connectToDB(db_entry)
    createGlobalCache(db_entry)
    for axiom_ids in generateProblemsForType(type_declaration,
                                             db_entry,
                                             db,
                                             reverse):
        if remote:
            runBlitzProve.apply_async((prover, axiom_ids, db_entry))
```

```
        else:
            runBlitzProve(prover, axiom_ids, db_entry)

def typeCheckAxioms(db_en):
    db = connectToDB(db_en)
    createGlobalCache(db)
    for axiom in list(db['SUMO_axioms'].find()):
        print(axiom)
        runPersistentProve('leo2',{'axiom_ids': [axiom['id']]}, 'mongo')

def checkTypesFromTop(db_en, start=0, fallback='leo2'):
    """ Check just the subgraphs of maximal size """
    db = connectToDB(db_en)
    createGlobalCache(db_en)
    i = 0
    for td in list(db['SUMO_types'].find()):
        if i >= start:
            print(td)
            axiom_ids = [a['id'] for a in list(getAxiomsWithType(td['declared_type']['name'], '
                mongo', db))]
            try:
                print(td['declared_type']['name'], axiom_ids, sep='\n')
                runBlitzProve('nitpick', {'axiom_ids': axiom_ids}, 'mongo')
            except:
                print('An error occured, running fallback: ', fallback )
                runBlitzProve(fallback, {'axiom_ids': axiom_ids}, 'mongo')
        i+=1

def dumpResultToFile(entry, field, filename):
    """ Assuming that entry is an dictionary extract the value of an key into a file.
    This is useful for extracting problems from the database for further validation."""
    with open(TMPDIR+filename,mode='w+') as f:
        f.write(entry[field])

def filenameGenerator(pattern, suf):
    """ Given a pattern, return a generator with return the pattern with increasing  numbers
        appended to it:
    pattern1, pattern2 ..."""
    i = 0
    while True:
        i += 1
        yield ''.join([pattern,str(i),suf])

def sieve(l, filters):
    """ Given a list of filter functions return the filtered list. This is just
    syntactic sugar. """
    return (e for e in l if all(f(e) for f in filters))

def dumpResults(db, db_name, pattern, suf, field, filters):
    """ Dump results from database into files for further experimentation """
    db = connectToDB(db)
    fg = filenameGenerator(pattern, suf)
    l = db[db_name].find()
    for e in sieve(l,filters):
        dumpResultToFile(e, field, next(fg))

def startJobOnTorque(prover, axiom_ids):
    """Creates a job on torque for prover and axiom_ids and returns a
handle for the task scheduler"""
    problem = createTHFStringFromProblemSet(getTypes({'axiom_ids':axiom_ids}),
                                            {'axiom_ids': axiom_ids}
                                            )
    jobname = writeProblemFileForTorque(problem)
    context = {
        'prover_command': '/'.join([os.getcwd(),provers[prover]['call'].__str__()]),
        'problemname': jobname,
        'file_path': '/'.join([TQJ_DIR[0:-1], jobname, jobname])
    }
    job = engine.render('test.sh', context)
    writeJobFileForTorque(job, jobname)
    jobid = sh.qsub('/'.join([TQJ_DIR[0:-1], jobname, 'job.sh']))
    return ('s', jobname, prover, axiom_ids, jobid)

def startMultiJobOnTorque(prover, axiom_idss):
    """Many small jobs put a large load on torque and can cause
crashes. Same as the previos function. Note that the handle is
slightly different"""
    problems = ( createTHFStringFromProblemSet(getTypes({'axiom_ids':axiom_ids}),
                                            {'axiom_ids': axiom_ids}
                                            ) for axiom_ids in axiom_idss )
    mjobname = random_string(25)
    while os.path.exists(TQJ_DIR+mjobname):
        mjobname = random_string(25)
    jobnames = writeMultiProblemFileForTorque(mjobname, problems)
```

```
    contexts = [{
        'prover_command': '/'.join([os.getcwd(),provers[prover]['call'].__str__()]),
        'problemname': mjobname,
        'file_path': '/'.join([TQJ_DIR[0:-1], mjobname, jobname])
    } for jobname in jobnames ]
    job = engine.render('m.sh', contexts[0])
    jobs = (engine.render('m1.sh', context) for context in contexts[1:])
    s = "\n echo '[@DIVIDER@]'\n".join(jobs)
    problemfile = job + "\n echo '[@DIVIDER@]'\n" + s + "\ntouch DONE\n"
    writeJobFileForTorque(problemfile, mjobname)
    # On long running jobs sometimes memory allocation of the cluster fails
    # causing the whole run to be in vain
    while True:
        try:
            jobid = sh.qsub('/'.join([TQJ_DIR[0:-1], mjobname, 'job.sh']))
            break
        except:
            print('Job start failed. Retrying in 15 seconds')
            time.sleep(15)
    return ('m' ,mjobname, jobnames, axiom_idss, prover, jobid)

def parseMultipleTorqueResults(mjob, prover, db_entry):
    """ Given an multiple job tuple determine whether an error occured somewhere during
        checking """
    db = connectToDB('mongo')
    with open('/'.join([TQJ_DIR[0:-1], mjob[1], 'job.out'])) as data:
        out = data.read()
        l = DIVIDER.split(out)
        for i in range(len(mjob[2])):
            try:
                prover_result = True if len(provers[prover]['satout'].findall(l[i]))>0 else
                    False
            except:
                prover_result = False
            if not prover_result:
                print(mjob[3][i],l[i], prover_result)
                db['SUMO_torque'].insert({'axiom_ids': mjob[3][i],
                                          prover+'_output': l[i],
                                          prover+'_result': prover_result,
                                          'further_check': True})



def isTorqueJobDone(jobname):
    return os.path.exists('/'.join([TQJ_DIR[0:-1], jobname, 'DONE']))

def parseTorqueResults(job, prover, db_entry):
    db = connectToDB(db_entry)
    with open('/'.join([TQJ_DIR[0:-1], job[0], 'job.out'])) as data:
        out = data.read()
        prover_result = True if len(provers[prover]['satout'].findall(out))>0 else False
        if not prover_result:
            db['SUMO_torque'].insert({'axiom_ids': job[1],
                                      prover+'_output': out,
                                      prover+'_result': prover_result})
        return prover_result

def TorqueJobTimeOut(job):
    """ Detect job timeouts """
    jobid = job[5] if job[0] == 'm' else job[4]
    return jobid in sh.qstat()

def checkAxiomsOnTorqueBySubSets(td, db_entry, prover='nitpick', reverse=False, MAX_JOBS=20,
      JOB_CHUNK=150):
    db = connectToDB(db_entry)
    joblist = []
    done_jobs = []
    current_jobs = 0
    problems = generateProblemsForType(td, db_entry, db, reverse)
    checked = 0
    to_check = 2**getTypeImportance(td, db_entry)-1
    done = False
    job_package = []
    while not done:
        # Some adaption to the work load on the cluster
        # However due to restriction concerning my filesystem quota this is rather limited
        if len(joblist) == 0:
            JOB_CHUNK += 10
        if len(joblist) == (MAX_JOBS-1):
            JOB_CHUNK = 150
        while len(joblist) < MAX_JOBS:
            l = list(i.islice(problems, JOB_CHUNK))
            if l != []:
                l = [ax['axiom_ids'] for ax in l]
```

```
                joblist.append(startMultiJobOnTorque(prover, 1))
            else:
                f = list(db['SUMO_torque'].find({'further_check':True}))
                if f != []:
                    for e in f:
                        joblist.append(startJobOnTorque('satallax',
                                                        e['axiom_ids']))
                        db['SUMO_torque'].update({'axiom_ids': e['axiom_ids']},
                                                 {'$set':
                                                  {'further_check': False}
                                                 })
        done_jobs = [job for job in joblist if isTorqueJobDone(job[1]) or
                     TorqueJobTimeOut(job)]
        joblist = [job for job in joblist if job not in done_jobs]
        for job in done_jobs:
            # compound job
            if(job[0]=='m'):
                parseMultipleTorqueResults(job, job[4], db_entry)
                checked += len(job[2])
                removeProblemFromTorque(job[1])
            if(job[0]=='s'):
                parseTorqueResults(job[1], job[2])
        print(td, checked, to_check)
        if joblist == []:
            done = True
        else : time.sleep(30)
```

The different prover modules starting with *nitpick*.

```
import re
import sh

nitpick = sh.Command('isabelle/bin/isabelle')
parameters = "600"
SATISFIABLE = re.compile('.*% SZS status Satisfiable.*')
FINITEMODELSTART =  re.compile('% SZS output start FiniteModel')
FINITEMODELEND = re.compile('% SZS output end FiniteModel')
```

Satallax

```
import re
import sh

satallax = sh.Command("satallax/bin/satallax.opt")
parameters = "-N"
SATISFIABLE = re.compile('.*% SZS status Satisfiable.*')
```

LEO-II

```
import re
import sh

leo2 = sh.Command('leo2/bin/leo.opt')
SATISFIABLE = re.compile('.*% SZS status Satisfiable.*')
```

The job templates for TORQUE. Single jobs on the cluster:

```
#PBS -N myJob
#PBS -d /home/bude/mziener/jobs/${ problemname }
#PBS -o job.out
#PBS -e job.err
#PBS -l nodes=1
#PBS -l pmem=1500mb
#PBS -l walltime=02:00:00
#PBS -M mziener@mi.fu-berlin.de

${ prover_command } ${ file_path }
touch DONE
```

The job files for compound jobs on the cluster.

```
#PBS -N myJob
#PBS -d /home/bude/mziener/jobs/${ problemname }
#PBS -o job.out
#PBS -e job.err
#PBS -l nodes=1
#PBS -l pmem=1500mb
#PBS -l walltime=04:00:00
#PBS -M mziener@mi.fu-berlin.de

${ prover_command } ${ file_path }
```

Each other new job is append with:

```
${ prover_command } ${ file_path }
```

List of Figures

References

[1] M. Heidegger, *Sein und Zeit.* Niemeyer, 1929.

[2] J. Álvez, P. Lucio, and G. Rigau, "Adimen-SUMO: Reengineering an Ontology for First-Order Reasoning."

[3] A. Pease and G. Sutcliffe, "First order reasoning on a large ontology," in *Proceedings of the CADE-21 Workshop on Empirically Successful Automated Reasoning in Large Theories*, vol. 257, 2007, pp. 59–69.

[4] C. Benzmüller and A. Pease, "Higher-order aspects and context in SUMO," *Journal of Web Semantics (Special Issue on Reasoning with context in the Semantic Web)*, vol. 12-13, pp. 104–117, 2012, dOI 10.1016/j.websem.2011.11.008.

[5] I. Niles and A. Pease, "Towards a standard upper ontology," in *Proceedings of the international conference on Formal Ontology in Information Systems-Volume 2001.* ACM, 2001, pp. 2–9.

[6] ——, "Linking lexicons and ontologies: Mapping wordnet to the suggested upper merged ontology," in *Proceedings of the 2003 internatial conference on Information and Knowledeg Engineering (IKE 03), LAS VEGAS*, 2003, pp. 412–416.

[7] M. Genesereth, R. E. Fikes, R. Brachman, T. Gruber, P. Hayes, R. Letsinger, V. Lifschitz, R. Macgregor, J. Mccarthy, P. Norvig, and R. Patil, "Knowledge interchange format version 3.0 reference manual," 1992.

[8] A. Pease and C. Benzmüller, "Sigma: An integrated development environment for formal ontology."

[9] C. Benzmüller, F. Rabe, and G. Sutcliffe, "Thf0 – the core tptp language for classical higher-order logic."

[10] G. Boole, *The Mathematical analysis of Logic.* Philosophical Library, 1847.

[11] A. De Morgan, *Formal Logic: or, the calculus of inference, necessary and probable.* Taylor and Walton, 1847.

[12] W. C. Kneale and M. Kneale, *The Development of Logic.* Oxford Eng., Clarendon Press, 1962.

[13] G. Frege, *Begriffsschrift, eine der arithmetischen nachgebildete Formelsprache des reinen Denkens.* L. Nebert Halle, 1879.

[14] J. E. Collins, *A History of the Theory of Types with Special Reference to Developments After the Second Edition of Principia Mathematica.* Open Access Dissertations and Theses. Paper 7211, 2005.

[15] E. Schröder, J. Lüroth, and K. E. Müller, *Vorlesungen über die Algebra der Logik: exakte Logik*. BG Teubner, 1890, vol. 1.

[16] B. Russell, "The Principles of Mathematics," *Bull. Amer. Math. Soc. 11 (1904), 74-93. DOI: 10.1090/S0002-9904-1904-01185-4 PII: S*, vol. 2, no. 9904, pp. 01185–4, 1904.

[17] A. N. Whitehead and B. Russell, *Principia mathematica*. University Press, 1912, vol. 2.

[18] L. Chwistek, "Über die Antinomien der Prinzipien der Mathematik," *Mathematische Zeitschrift*, vol. 14, no. 1, pp. 236–243, 1922.

[19] F. P. Ramsey, "The Foundations of Mathematics," *Proceedings of the London Mathematical Society*, vol. 2, no. 1, pp. 338–384, 1926.

[20] K. Gödel, "Über formal unentscheidbare Sätze der Principia Mathematica und verwandter Systeme i," *Monatshefte für Mathematik und Physik*, vol. 38, no. 1, pp. 173–198, 1931.

[21] A. Tarski, "The concept of truth in formalized languages," *Logic, semantics, metamathematics*, vol. 2, pp. 152–278, 1956.

[22] A. Church, "A formulation of the simple theory of types," *The journal of symbolic logic*, vol. 5, no. 2, pp. 56–68, 1940.

[23] G. Sutcliffe and C. Benzmüller, "Automated reasoning in higher-order logic using the TPTP THF infrastructure," *Journal of Formalized Reasoning*, vol. 3, no. 1, pp. 1–27, 2010.

[24] P. B. Andrews, *An Introduction to mathematical logic and type theory: to truth through proof*. Kluwer Academic Pub, 2002, vol. 27.

[25] W. M. Farmer, "The seven virtues of simple type theory," *Journal of Applied Logic*, vol. 6, no. 3, pp. 267–286, 2008.

[26] L. Henkin, "Completeness in the theory of types," *The Journal of Symbolic Logic*, vol. 15, no. 2, pp. 81–91, 1950.

[27] P. B. Andrews, "Classical type theory, handbook of automated reasoning," 2001.

[28] C. Benzmüller, F. Rabe, and G. Sutcliffe, "THF0–the core of the TPTP language for higher-order logic," in *Automated Reasoning*. Springer, 2008, pp. 491–506.

[29] F. Pfenning and C. Schürmann, "System description: Twelf—a meta-logical framework for deductive systems," in *Automated Deduction—CADE-16*. Springer, 1999, pp. 202–206.

[30] S. Vinoski, "Advanced message queuing protocol," *Internet Computing, IEEE*, vol. 10, no. 6, pp. 87–89, 2006.

[31] G. Staples, "Torque resource manager," in *Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, ser. SC '06. New York, NY, USA: ACM, 2006. [Online]. Available: http://doi.acm.org/10.1145/1188455.1188464

[32] T. Nipkow, L. C. Paulson, and M. Wenzel, *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, ser. LNCS. Springer, 2002, vol. 2283.

[33] C. Benzmüller, L. C. Paulson, F. Theiss, and A. Fietzke, "Leo-ii-a cooperative automatic theorem prover for classical higher-order logic (system description)," in *Automated Reasoning*. Springer, 2008, pp. 162–170.

[34] S. Schulz, "E-a brainiac theorem prover," *AI Communications*, vol. 15, no. 2, pp. 111–126, 2002.

[35] C. E. Brown, "Satallax: An automatic higher-order prover," in *Automated Reasoning*. Springer, 2012, pp. 111–117.

[36] N. Sorensson and N. Een, "Minisat v1. 13-a sat solver with conflict-clause minimization," *SAT*, vol. 2005, p. 53, 2005.

[37] G. Sutcliffe, "The szs ontologies for automated reasoning software."

[38] J. D. Hunter, "Matplotlib: A 2d graphics environment," *Computing In Science & Engineering*, vol. 9, no. 3, pp. 90–95, 2007.

[39] B. Chandrasekaran, J. R. Josephson, and V. R. Benjamins, "What are ontologies, and why do we need them?" *Intelligent Systems and Their Applications, IEEE*, vol. 14, no. 1, pp. 20–26, 1999.