TEMPLAR

Efficient Determination of Relevant Axioms in Big Formula Sets for Theorem Proving

---

Diplomarbeit

von

Mario Frank

Universität Potsdam
Institut für Informatik
Professur Theoretische Informatik

Aufgabenstellung und Betreuung:
Prof. Dr. Christoph Kreitz
Prof. Dr. Torsten Schaub

Potsdam, den 20. Juni 2013

Frank, Mario
Mario.Frank@uni-potsdam.de
TEMPLAR
Diplomarbeit, Institut für Informatik
Universität Potsdam, Juni 2013

Published online at the
Institutional Repository of the University of Potsdam:
URL http://opus.kobv.de/ubp/volltexte/2014/7211/
URN urn:nbn:de:kobv:517-opus-72112
http://nbn-resolving.de/urn:nbn:de:kobv:517-opus-72112

Thanks to Prof. Dr. Kreitz for the great supervision, to Jens Otten for the countless discussions on calculi and transformations, Tim Richter for providing access and support to the hardware on which TEMPLAR was exhaustedly tested and for restarting it after TEMPLAR consumed all memory and destabilized the system. Furthermore, I want to thank Prof. emerit. Dr. Wolfgang Bibel for providing me a version of his Deduction book which helped me to brighten my understandings for some special concepts.

Also, many thanks to Jörg Zinke, Stefan Gasterstädt and the Operating Systems and Distributed Systems chair for providing this incredible amazing LaTeX thesis template.

Finally, thanks to all those people who provided me coffee and Club Mate.

Selbständigkeitserklärung

Hiermit erkläre ich, daß ich die vorliegende Arbeit selbständig angefertigt, nicht anderweitig zu Prüfungszwecken vorgelegt und keine anderen als die angegebenen Hilfsmittel verwendet habe. Sämtliche wissentlich verwendeten Textausschnitte, Zitate oder Inhalte anderer Verfasser wurden ausdrücklich als solche gekennzeichnet.

Potsdam, den 20. Juni 2013

_____

Mario Frank

Abstract

This document presents an formula selection system for classical first order theorem proving based on the relevance of formulae for the proof of a conjecture. It is based on unifiability of predicates and is also able to use a linguistic approach for the selection. The scope of the technique is the reduction of the set of formulae and the increase of the amount of provable conjectures in a given time. Since the technique generates a subset of the formula set, it can be used as a preprocessor for automated theorem proving. The document contains the conception, implementation and evaluation of both selection concepts. While the one concept generates a search graph over the negation normal forms of the given formulae, the linguistic concept analyses the formulae and determines frequencies of lexemes and uses a tf-idf weighting algorithm to determine the relevance of the formulae. The system was also evaluated at the world championship of the automated theorem provers (CASC-24) and the evaluation of the results of the CASC and the benchmarks with the problems of the CASC of the year 2012 (CASC-J6) show that the concept of the system has positive impact to the performance of automated theorem provers. Also, the benchmarks with two different theorem provers which use different inference calculi have shown that the selection is independent from the calculus.

Zusammenfassung

Dieses Dokument stellt ein System vor, das aus einer (großen) gegebenen Menge von Formeln der klassischen Prädikatenlogik eine Teilmenge auswählt, die für den Beweis einer logischen Formel relevant sind. Ziel des Systems ist, die Beweisbarkeit von Formeln in einer festen Zeitschranke zu ermöglichen oder die Beweissuche durch die eingeschränkte Formelmenge zu beschleunigen. Das Dokument beschreibt die Konzeption, Implementierung und Evaluation des Systems und geht dabei auf die zwei verschiedenen Ansätze zur Auswahl ein. Während das eine Konzept eine Graphensuche auf den Negations-Normalformen der Formeln durchführt, indem Pfade von einer Formel zu einer anderen durch Unifikation von Prädikaten gebildet werden, analysiert das andere Konzept die Häufigkeiten von Lexemen und bildet einen Relevanzwert durch Anwendung des in der Computerlinguistik bekannten tf-idf-Maßes. Es werden die Ergebnisse der Weltmeisterschaft der automatischen Theorembeweiser (CASC-24) vorgestellt und der Effekt des Systems für die Beweissuche analysiert. Weiterhin werden die Ergebnisse der Tests des Systems auf den Problemen der Weltmeisterschaft aus dem Jahre 2012 (CASC-J6) vorgestellt. Es wird darauf basierend evaluiert, inwieweit die Einschränkungen die Theorembeweiser bei dem Beweis komplexer Probleme unterstützen. Letztendlich wird gezeigt, dass das System einerseits positive Effekte für die Theorembeweiser hat und andererseits unabhängig von dem Kalkül ist, das die Theorembeweiser nutzen.

Contents

1. Introduction

Automated deduction is not just an academic field of research nowadays. Multiple concepts are used in industrial software and hardware verification. For example, the transformation of models for Electronic Stability Control (ESC)[1] into source code (e.g. C code) can be verified by use of automated deduction.

But the systems which can check the validity of such transformations need to be able to handle arithmetic, type systems and big formula sets, too since physical aspects can contain many formulae. Some of the systems with these properties work fully automated and are called automated theorem provers. They get a set of formulae which are seen as valid (like the laws of mechanics, called axioms) and a conjecture which is a formula that has to be proven with the laws.

Since the theorem provers need to be as fast as possible but get increasing sets of formulae, they sometimes suffer from too big sets which lead to slow proof attempts. Quite often, they can not prove some conjecture due to the huge amount of formulae or they just need too much memory.

But theorem provers need to analyse all given formulae together with the conjecture, transform them into specific normal forms and proof the conjecture with the rules of a calculus. And the bigger the number of axioms becomes, the more inference steps are possible which enormously increases the memory usage and calculation time.

In practice, not all axioms are needed to prove the conjecture since there can be many laws which are not even relevant for the conjecture or some inference step. Thus, using all of the formulae would be a waste of resources. This is where relevance filter can be applied. They have the aim to decide about the relevance of a set of formulae and select the ones which can be seen as relevant enough.

The current state-of-the-art filtering technique is described in **SinE** [HV11] and used by well-known theorem proves like **E** [Sch04], **Vampire** [HKV11], **iProver**[Kor08], **MaLARea** [USPV08] and **E-KRHyper** [PW07].

Since theorem provers need to be tested, some of them attend the annual CADE ATP Systems Competition (CASC) in order to compete with each other. Some are specialised on arithmetics or first order logic and some are specialised on finding non-theorems but they all can benefit from filtering techniques. There are many different terms used to describe filtering techniques. In order to use a consistent description, this document uses the term "relevance filter" or pruner since usually, all techniques use a relevance metric in order to filter(prune or reorder) the formula sets.

In the CASC J6[Sut12b], the competition which took place in Manchester in June 2012, all theorem provers of the Large Theorem Batch (LTB) division used the SinE filtering technique

[1] This is a module in vehicles which aims to stabilize vehicles. The loss of traction is detected and reduced by automatically applying break actions. For reference, see the patent, for example: https://register.dpma.de/DPMAregister/marke/register/2912578/DE

while leanCoP-ARDE used unification based relevance filtering. leanCoP[Ott08]/[OB03] is a lean connection calculus based theorem prover, implemented by Jens Otten.

This document addresses the TEMpestuous Pruner based on Logical Axiom Relevance (TEMPLAR) which contains the extension of the filtering technique of ARDE, which was partly described and published in [Fra12b].

The TEMPLAR is a modular system which has the scope to select relevant formulae for the proof of a formula and can use both logical and also linguistic relevance approaches. The name is derived from the Knights Templars[2] and has the aim to describe the structure of the system. Templars are well-armed and fight with stormy attacks against their foes and are merciful to the innocent. And TEMPLAR just does the same. It uses tempestuous attacks on the sets of formulae to reduce them (to prune them) but is merciful to those formulae which he accepts as innocent (relevant). And he does this mostly with logical relevance which is based on unification algorithms. But there are many heuristics to prune the search space and there is even an implementation of a linguistic concept.

In this years CASC (CASC-24[3]), a beta version of TEMPLAR with a reduced set of heuristics (in order to save memory) in combination with leanCoP named *TEMPLAR::leanCoP* with version 0.8 was handed in in order to compare the new filtering approach and the effects with the other theorem provers.

This document is separated in six parts where the first one describes the preliminaries which are mostly well-known but also contain some linguistic aspects. The second part described the concept of TEMPLAR including the search strategies, the formula analysis and import and also some structure sharing approaches. The third part describes the most important aspects of the implementation including the relevance filtering, search graph restrictions and output mechanisms. The evaluation chapter contains results from tests with TEMPLAR on a set of problems which were used in CASC J6 and the related work chapter describes some concepts which have common aspects with TEMPLAR. The last part gives a brief conclusion on the effects of TEMPLAR and some possible improvements which can be done. The appendix contains all detailed benchmarks with TEMPLAR together with leanCoP and the eprover(E) and also some usage information for TEMPLAR including the installation.

[2] An ancient fellowship of Christian soldiers. There are many references to them so a quite new academic one was chosen[Ger96]

[3] http://www.cs.miami.edu/~tptp/CASC/24/

2. Preliminaries

This chapter covers knowledge which is partly common for computer scientists and partly out of scope. But since the terms which are used in this document need to be clear, all needed knowledge is introduced.

The covered topics are the syntax of classical first order logic, normal form transformations, unification and connections(all included in section 2.1), search concepts in graphs and graph reductions (included in section 2.2) and also linguistic approaches to automated text summarization (included in section 2.3).

All introduced concepts can be found in primary literature for the specific topics but references to quite clear descriptions or initial authors of the concepts are given in the specific parts.

2.1 Fundamentals of Classical First Order Logic and Reasoning

This section copes with the fundamentals of classical first order logic (fol) including normal form transformations, unification and connections. All these topics are described in-depth in [Bib92] and [BBJ07] and many other literature contains these topics, too.

First Order Logic and Normal Forms In order to use consistent terms, the syntax of classical first order logic formulae needs to be introduced.

Definition 1 (Syntax of first order logic) A formula in first order logic consists of predicates (e.g. $p(x,Y)$) and equalities (e.g. $x = y$ or $equal(x,y)$) which are bound by the logical junctions negation ($\neg$), conjunction ($\wedge$), disjunction ($\vee$), implication ($\rightarrow$), equivalence ($\leftrightarrow$), existential quantification ($\exists$) and the universal quantification ($\forall$). Predicates of a higher arity then zero and equalities contain terms which are variables (e.g. Z), functions (e.g. $f(x)$) and constants (functions with zero arity, e.g. x). All variables that are not free, are either bound by an existential quantification (e.g. $\exists X,Y : X = Y$) or by an universal quantification (e.g. $\forall X : p(x)$). □

> **Note:** Technically, equalities can be seen as special predicates with a fixed arity of two and symmetry property concerning unification. Thus, the terms predicate or (equality-)predicate will be used also as description for equalities whenever no distinction between equality and predicate is needed.

The figure 2.1, for example shows a set of first order formulae conforming to the syntax definition which contains a conjecture (Conj) which needs to be proven with the formulae (Ax 1 - Ax 6).

There are some fact formulae which state that tweety is a bird (Ax 3) but not a penguin (Ax 5) and that ralph and tux both are penguins (Ax 6, Ax 4). Also, there are quantified formulae which state that every bird which is not a penguin is able to fly (Ax 1) and that there is a substitution for the variable X such that X is able to fly if and only if (iff) X is a bird and has wings (Ax 2).

But to simplify the automated reasoning with the formulae, usually they are transformed into normal forms like the negation normal form, Skolem normal form or clausal normal form. In order to transform a formula into clausal normal form, it already needs to be in Skolem normal form and the Skolem normal form transformation works only on formulae which are in prenex normal form.

Thus, all formulae are first transformed into negation normal form, then into prenex normal form and Skolem normal form and finally into clausal normal forms.

Conj) $canFly(tweety)$
Ax 1) $\forall X : ((bird(X) \wedge \neg penguin(X)) \Rightarrow canFly(X))$
Ax 2) $\exists X : (canFly(X) \Leftrightarrow (hasWings(X) \wedge bird(X)))$
Ax 3) $bird(tweety)$
Ax 4) $penguin(tux)$
Ax 5) $\neg penguin(tweety)$
Ax 6) $penguin(ralph)$

Figure 2.1: Formulation of the Tweety-problem as first order logic formulae

The negation normal form is a simplification which only includes the logical junctions negation, disjunction, conjunction and quantifiers. The implication and equivalence are replaced by semantically equivalent formulae and all negations are propagated into sub-formulae until they are located directly before the predicates and equalities.

The rules for semantic equivalence over some arbitrary formulae A and B are (including De Morgan's laws):

- $A \Rightarrow B \equiv \neg A \vee B$ (Elimination of implication)
- $A \Leftrightarrow B \equiv (\neg A \vee B) \wedge (A \vee \neg B)$ (Elimination of equivalence)
- $\neg \forall A(F) \equiv \exists A(\neg F)$ (Negation of universal quantification)
- $\neg \exists A(F) \equiv \forall A(\neg F)$ (Negation of existential quantification)
- $\neg(A \wedge B) \equiv (\neg A \vee \neg B)$ (De Morgan's law 1)
- $\neg(A \vee B) \equiv (\neg A \wedge \neg B)$ (De Morgan's law 2)
- $\neg\neg A \equiv A$ (Elimination of double negation)

The fact whether a predicate is negated in negation normal form or not can be described as it's polarity. All predicates in the negation normal form of the formula get a polarity which is negative ("-") if the predicate is negated and positive ("+"), otherwise. There are different notions for polarities like T for positive and F for negative as used in [Smu95], for example.

After transforming a formula to negation normal form, it can be transformed into prenex normal form and Skolem normal form. The prenex normal form is a form where no ambiguities about the variable bindings are present.

Definition 2 (rectified prenex normal form) A formula in classical first order logic is **rectified** if no two quantifiers bind the same variable with the one quantifier being in the scope of the other and no variable is unbound. A formula is in prenex normal form, iff it has the form $Q_1X_1...Q_nX_n(F)$ where $Q_i \in \{\exists, \forall\}$ with X_i being a variable and $i \in \mathbb{N}$. Furthermore, F is a formula which does not contain any quantifiers and is called the **matrix** while the quantifier sequence is called **prefix**. □

Note: Technically, a formula can be seen as in rectified prenex normal form if no variables are unbound and all ambiguities by nested quantifications of the same variable are resolved by renaming the bound variable in the nested quantifier and in the sub-formulae which are contained in the nested quantified formula.

Every formula can be transformed into prenex normal form if it is not already in this form which was proven in [BBJ07]. The transformation is done by shifting all quantifiers to the left (the outer scope) end and replacing all variables which would become bound by them with a new variable. All unbound variables are bound by introduction a new universal quantifier in the most outer scope.

Definition 3 (Skolem normal form) A formula in classical first order logic which is in rectified prenex normal form is in Skolem normal form iff the only quantifiers are universal quantifiers. □

The fact that every fol-formula has a Skolem normal form which also is satisfiability-equivalent[1] ($\equiv_{sat}$) to the original formula was proven in [BEL01] for example.

The transformation into Skolem normal form is done by replacing every existentially quantified variable by a Skolem function (or constant) which does not occur in the matrix and also not in any other formula. All universally quantified variables left to the existentially quantified variable form the arguments of the new function.

To illustrate this, consider the following two first order formulae in figure 2.2. The skolem-

$$\forall X \exists Y : (contains(X,Y)) \equiv_{sat} \forall X(contains(X, sk\_fun1(X)))$$
$$\exists X : (isNaturalNumber(X)) \equiv_{sat} isNaturalNumber(sk\_fun1)$$

Figure 2.2: Skolemization of fol-formulae

ization of the first formula is done by replacing the variable Y with sk_fun1 and defining X as its argument. The skolemization of the second formula is done by replacing the variable X with sk_fun1 which is a Skolem constant.

Since most of the theorem provers use a clause normal form or the definitional normal form for the proof search and TEMPLAR uses meta-informations from this forms, they will be introduced,too.

Definition 4 (Clause normal form) A fol-formula is in clause normal form, if it is in Skolem normal form and a conjunction of disjunctions (**conjunctive normal form**) or a disjunction of conjunctions (**disjunctive normal form**). For the conjunctive normal form, the disjunctions are clauses and contain only predicates (and equalities). For the disjunctive normal form, the clauses are conjunctions and contain only predicates (and equalities). □

Every first order formula can be transformed into a clause normal form which is obviously possible by just applying the laws of distributivity.

While the naive transformation via the laws of distributivity can lead to exponential growth of the formula, the definitional transformation only leads to linear growth in size of the original formula as noticed in [Ede92]. The definitional transformation is done by introducing a new name (definition) for every sub-formula of the Skolem normal form. By this, the formula is virtually flattened. Every definition contains informations about the contained variables in this sub-formula.

[1] The skolemized formula is satisfiable iff the original formula is satisfiable and unsatisfiable iff the original formula is unsatisfiable.

Unification and Connections Since the concept of TEMPLAR is mostly based on unifications and connections, these concepts also need to be introduced and can also be found in standard literature like [Bib92].

Definition 5 (Substitution and Unifiability) Let $\mathcal{V} = \{V_1, V_2, ..., V_n\}$ with $n \in \mathbb{N}$ be a set of variables. Also, let $\mathcal{T} = \{T_1, T_2, ..., T_n\}$ with $n \in \mathbb{N}$ be a set of terms (variables, functions and constants). Then, the mapping $\sigma : \mathcal{V} \to \mathcal{T}$ from variables to terms is called a **substitution**.

Two predicates $p(s_1, s_2, ..., s_n)$ and $p(t_1, t_2, ..., t_n)$, with s_i, t_i being terms with $i \in \mathbb{N}$ are **unifiable** iff there is a substitution such that $\sigma(s_i) = \sigma(t_i)$ holds for every $i \in \mathbb{N}$. □

If the predicates are unifiable, then σ is called their unifier and σ is called the most general unifier (mgu), if an unifier θ exists for every other unifier σ' such that $\sigma' = \theta(\sigma)$. Precisely, every unifier is constructible by applying a substitution to the most general unifier.

But substitutions of the form $\sigma(Y) = f(Y)$, for example would lead to infinite loops in the unification when not being avoided. Thus, unification algorithms usually implement an **occurs-check**. This check fails, if a function would substitute a variable while containing it. The unification fails if the occurs check fails since no finite set of substitutions would be constructible.

With unifications being defined, the connections which are the paths in the graph search of TEMPLAR, can be described.

Definition 6 (Connection) A pair $\{p, \neg p\}$ of predicates which are unifiable is called a **connection**. A pair $\{A = B, \neg(C = D)\}$ of equalities with A, B, C and D being terms is a connection iff $A = B$ is unifiable either with $\neg(C = D)$ or with $\neg(D = C)$. □

In the following section, a brief introduction into graph theory will be given to make the synopsis of used terms in the scope of this document consistent.

2.2 Fundamentals of Graphs and Graph Search Algorithms

In this short section, we will take a look at some basics about graphs and search algorithm types which can also be found in [Sip96].

Definition 7 (Graph) A graph $G = (N, E)$ consists of a set N of nodes(vertices) and a set E of edges with every element of the set of edges being a pair $e = (n_i, n_j)$ where $i, j \in \mathbb{N}$ and $n_i, n_j \in N$.

A directed graph follows the rule that every edge is directed such that the pair e is an edge with n_i being the source and n_j being the target. For undirected graphs, n_i and n_j are both source and target.

A graph is acyclic if there is no path $p = (n_i, n_{i+1}, ..., n_{i+m})$ with $m, i \in \mathbb{N}$ such that $n_i = n_{i+m}$. □

There are two different commonly used graph search types, **depth-first** and **breadth-first**. The (standard) depth-first search always extends the leftmost path until it cannot be extended and then tries to extend the next leftmost path. By this, the depth-first search always finds (if it terminates) the first solution but not necessarily the shortest path. The (standard) breadth-first search tries to extend every path in every iteration of the search until the searched-for object is found. Due to this, the breadth-first search always finds the shortest path. While the depth-first search can run into cycles and thus does not necessarily terminate, the breadth-first search always terminates if the searched-for object can be found in the graph even if there are cycles.

Sometimes, a node n in a graph is a target node of directed edges of different source nodes n_i with $i \in \mathbb{N}$. Then, for every node n_i with the target n, the sub-graph of n is identical and n is called a **nexus**. This is described in detail in [Bir10] and the nexus concept can be adapted in multiple ways in automated reasoning.

For example, if two predicates both contain a function (e.g. $f(x,1,3)$), the function root f in the graph representation of the predicates would be a nexus which states that the complete function graph can be shared by the two predicates. Also, if two connections have the same target predicate of the same formula, further outgoing connections from the shared target predicate will be identical in both paths if the substitutions of both connections concerning the target formula are unifiable and the formula was visited in both paths for the first time.

The next section covers some basic linguistic aspect commonly used in the computational linguistics in Natural Language Processing (NLP).

2.3 Selected Linguistic Aspects

In computational linguistics, natural language like speech, documents and other media are analysed in order to compute specific relations like dominance of a word over another word or a sequence of words. Commonly, words ("the","bear",...) and punctuation (",", ".",...) are called token. Multiple token can be joined into spans or structures (e.g. "the bear" and "the wolf is howling"). While spans merely overlap token, structures may overlap both spans and structures. This constructs make it possible to recognise (though with potential failures) the meaning of sentences and documents in order to automatically produce summarizations of texts. Also, computational linguistic concepts can be used to decide whether a subset of a set of documents address the same topic.

But in order to do these things, the language has to be analysed. For example, to recognise the topic of a text, all relevant words or sequences of words need to be found. One of the metrics used for the topic recognition is the tf-idf value.

tf-idf-value In linguistics, the frequency of a word in one document (text) is called **text frequency**(tf) and the frequency of a word in different documents is called **document frequency** (df). For the df, if a word is contained in a document multiple times, it is just counted once.

The tf-idf-value (probably originated from [Jon72]) of a word in a specific text is used to decide, how relevant a word is for the text and idf is the **inverse document frequency**. The text frequency is multiplied with the inverse document frequency. A high frequency of a word in one text may increase the tf-idf-value and a high document frequency may lower it. The aim is to penalise so-called common word like "it", "the" and "I". The probability that the word "the" will occur in nearly every English document is high but the probability that "vehicle" will often occur is quite low unless most of the documents have some vehicles as topic.

We will now consider the formal definitions of the text frequency and the inverse document frequency.

Definition 8 (Text Frequency) The text frequency $tf(w,d)$ if the frequency of the word w in document d. It can be calculated as

- boolean text frequency :
  $tf(w,d) = \begin{cases} 1 & w \in d \\ 0 & w \notin d \end{cases}$
- normalised text frequency :
  $tf(w,d) = \frac{f(w,d)}{max\{f(v,d)|v \in d\}}$
  where f is the raw frequency

. While the boolean frequency merely tells whether the word is contained, the normalised frequency establishes a relation between the frequency of the word w and the most common word in the document. □

The inverse document frequency is a measure of the commonness of a word in the set of documents.

Definition 9 (Inverse Document Frequency) Let $D = \{d_1, d_2, ..., d_n\}$ with $n \in \mathbb{N}$ be a set of documents and w a word for which we want to know the inverse document frequency. Then, the inverse document frequency is defined as followed.
$idf(w,D) = log \frac{|D|}{|\{d \in D|w \in d\}|}$ □

The more common a word is in the documents, the smaller the inverse document frequency will be. Due to this, multiplying the inverse document frequency of an extremely common word with the text frequency usually leads to a smaller tf-idf-value. Thus, a high tf-idf-value for a word indicates that this word is quite uncommon in all documents but common in the one document, for example.

3. Concept

This concept addresses the selection of first order formulae from big formula sets in order to assist theorem provers which can not handle those big formula sets. The selected formulae need to be relevant for the proof of a specified formula. But for this, the assisting system needs to decide about the relevance. In order to be able to decide about the relevance of some formulae and select the ones which are classified as relevant we need to import and represent these formulae in some machine readable form. Thus, we need some input format which can specify first order logic formulae and can be processed easily.

One of this formats is the TPTP[Sut09] syntax which specifies a generic syntax definition which can be used to represent logic formulae of different types. One of this specified languages is the first order formula (fof)-language which is used in this concept. The formulae in fof-syntax need to be imported into internal data structures of the system TEMPLAR. The import and representation process will be addressed in the next section. Following to this topic, the analysis of the formulae and generation of some metrics which can aid the theorem prover or the selection engine will be described in the section 3.2.

After importing and analysing the formulae, the concrete selection of relevant formulae needs to be done which will be the subject of the sections 3.3 and 3.4. While the section 3.3 covers the unification based algorithms, the section 3.4 describes some linguistic approach which was partially used in some other systems.

3.1 Data Import and Representation

Now, the import and representation concept will be described in order to give a brief overview. This overview covers the original form of the formulae, the negation normal form and specific information about the Thousands of Problems for Theorem Provers (TPTP) syntax. All conventions of the TPTP syntax can be found in the technical report [1] by Geoff Sutcliffe. The formulae are held persistent in files conforming to the TPTP syntax. The file itself can contain an arbitrary number of formulae which are classified as different types of first order logic formulae. The file also contains an arbitrary number of import statements which are references to other TPTP files which contain more formulae to import. This makes a domain specific fragmentation of formula sets possible since it would not be adequate to mix first order formulae with propositional logic, for example.

Each formula in TPTP syntax has one of the following forms

```
fof(NAME,TYPE,FORMULA,ANNOTATIONS).
fof(NAME,TYPE,FORMULA).
```

where NAME is the TPTP-internal name of the formula. The field TYPE is a classification about the validity or purpose of the formula, FORMULA is the raw logic formula in TPTP syntax and the optional ANNOTATIONS field contains information about the source of the formula, for example.

Table 3.1 summarizes the most important types of TPTP formulae. The base type is the type **axiom** which resembles formulae which are generally valid. Formulae of the type **conjecture** or **question** are formulae whose validity needs to be proven. For formulae of the type question, the variable substitutions for the most outer existentially qualified variables need to be specified by the theorem prover after successfully finding a proof. Formulae of the types **lemma** and **theorem**

[1] http://www.cs.miami.edu/~tptp/TPTP/TR/TPTPTR.shtml

can be seen as valid but are redundant with respect to the axioms in the axiom set. If a lemma or theorem is not redundant, the axiom set is ill-formed and thus inconsistent. Formulae of the types **hypothesis** and **assumption** also can be seen as valid but are not necessarily valid for all formula sets.

TPTP formula type	Valid	Special
conjecture/question	unknown	output yes/no and variable bindings for question
axiom	yes	
lemma,theorem	yes	must be redundant w.r.t. axioms
hypothesis	yes	
assumption	yes	

Table 3.1: Most important formula types in the TPTP library and their properties

These TPTP files are read and all formulae are imported into a graph representation which can map the hierarchy of sub-formulae and terms of predicates. After that, all formulae are transformed into negation normal form and Skolem normal form. While conjectures and questions are seen as invalid and thus negated (polarised negative) before the transformation, all other formula types are assumed to be valid and thus not negated (polarised positive). This polarisation simulates the negation of the logical validity which states that the conjecture is the logical consequence of the set of axiom like formulae. Negating this, the negated conjecture is no logical consequence and thus, there is a possibility to show that there is a way to show that the negated conjecture is refutable with respect to the set of axioms.

Since every formula is a hierarchic structure (a directed graph), we need some terms to consistently describe the structure. Predicates, equalities, disjunctions and conjunctions are called formulae hereafter and equalities and predicates are summarised as connectible formulae. The TPTP structure of the formula will be called root formula since it contains the information of it's name, type and the actual formula structure and after the import the negation normal form formula, too.

Formula Representation Every imported formula is represented in two different forms - original formula structure and negation (skolemized) normal form. The TPTP specifies the logical operations negation ($\sim$), disjunction (;), conjunction (&), negated disjunction (NOR, $\sim$;), negated conjunction (NAND, $\sim$ &), left oriented implication ($<=$), right oriented implication ($=>$), equivalence ($<=>$), negated equivalence ($<\sim>$), equality ($=$), negated equality ($!=$), existential quantification (?) and universal quantification (!). Those operators are all supported by the concept of TEMpestuous Pruner based on Logical Axiom Relevance (TEMPLAR).

In table 3.2, the most important formula types and their representation in the TPTP-syntax and their mapping to negation normal form are summarized. Commonly, equivalence is first transformed into a conjunction of implications and then into a conjunction of disjunctive formulae. But the intermediate transformation can be and is omitted. Negations are propagated to sub-formulae as in the standard negation normal form transformation with use of De Morgan's laws, if necessary.

The identifiers X and Y which are used in the table are place-holders for formulae or for terms in case of inequality.

The existential and universal quantifiers are removed during the (optional) skolemization. After

Formula type	Original(TPTP input)	Negation Normal Form
Inequality	$X!=Y$	$\sim(X=Y)$
Equivalence	$X<=>Y$	$(\sim X;Y)\&(X;\sim Y)$
Negated equivalence	$X<\sim>Y$	$(X\&\sim Y);(\sim X\&Y)$
Left-implicative	$X<=Y$	$X;\sim Y$
Right-implicative	$X=>Y$	$\sim X;Y$
NAND	$X\sim\&Y$	$\sim(X\&Y)$
NOR	$X\sim;Y$	$\sim(X;Y)$

Table 3.2: Mapping of TPTP input formulae to negation normal form

the negation normal form transformation was done, every normal form formula has the following set of information:

- For all formulae: The containing **root formula** and the potentially empty set of recursively containing **super-formulae**
- For conjunction and disjunction: The roots of the **sub-formula** graphs
- For connectible formulae: The **arity** and the roots of the **term** graphs

Every function with an arity of at least one knows it's direct sub-terms in the sub-graph and optionally, which variables are contained in the sub-graph. Functions with an arity of zero,conforming to the TPTP syntax, are called constants and constants can be numbers, single-quoted strings or words which begin with a lower case letter.

Structure Sharing The formula and term hierarchies are called graphs and not trees. The reason is that the concept includes structure sharing [MB72] and the graphs are directed. Structure sharing is a concept where every object which occurs at different positions and is identical in every position can be seen as unique object which is shared between every occurrence. In the concept of TEMPLAR, restricted structure sharing is used in order to speed up the unification process. Every constant, number, single-quoted string and function classifier (name and arity) is unique and shared between all formulae contained in the formula set. Variables are shared in one formula since two variables which occur in different formulae could have the same name but need to be seen as different variables. Thus, calling the formulae trees would be formally wrong. They need to be seen as directed graphs with every super-formula or super-term being the source and every formula or term being a target since extended structure sharing. The restriction in the structure sharing is that two functions whose graphs are identical are not shared since this would require to make an identity check. The same holds for formulae which are syntactically identical. An example formula is the formula *equal_defn* from the problem set *SET002+3*[2] which is shown in negation normal form in figure 3.1 which is a visualization generated by TEMPLAR.

As it can be seen, the variables are only existent once but referenced by multiple predicates.

[2] http://www.cs.miami.edu/~tptp/cgi-bin/SeeTPTP?Category=Problems&Domain=SET&File=SET002+3.p

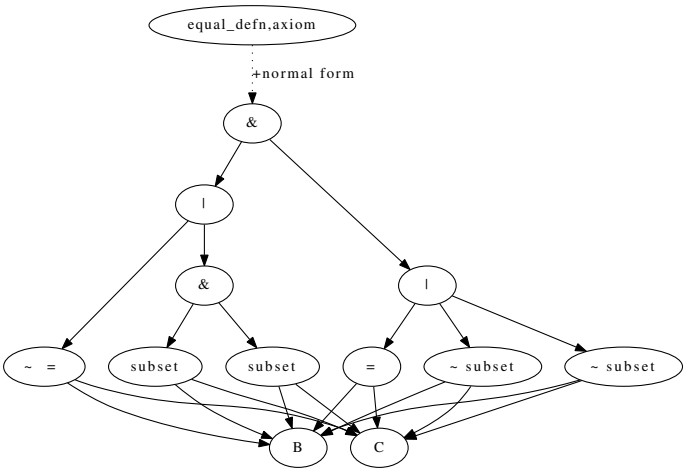

Figure 3.1: Formula in negation normal form with visualized structure sharing

3.2 Analysis

In this section we will take a look at some metrics which are generated by TEMPLAR during the negation normal form transformation or directly thereafter. These metrics include the worst-case clause count, the original formula depth, the structure containment information and the equality scopes.

Clause Count Theorem provers usually transform every formula into conjunctive normal form or disjunctive normal form which can lead to exponential growth of the clause count. Though some theorem provers are able to create a definitional normal form which does not yield exponential growth, the standard transformations may be more adequate for the proof search. To help the theorem prover to decide which transformation should be done, TEMPLAR can compute the worst case count of clauses which would be created in the standard transformation [dlT92] (conjunctive normal form) and give this information together with the selected formulae to the theorem prover. This informations are generated during the negation normal form transformation. Since TEMPLAR uses the negative representation of validity, a conjunctive normal form would be created. Thus, the predicates are distributed over disjunctions and the clause count can be calculated as described in table 3.3 where $|X|$ is the count of clauses which are generated by X.

Formula type	**clause count**
Conjunction	$\|subformula_1\| + \|subformula_2\| + ... + \|subformula_n\|$ for $n > 1$
Disjunction	$\|subformula_1\| * \|subformula_2\| * ... * \|subformula_n\|$ for $n > 1$
Predicate/Equality	1

Table 3.3: Clause Count Calculation Formulae

Thus, if a disjunction is generated by the negation normal form transformation, the multiplicative formula is used and the additive formula is used, if a conjunction is generated.

Formula Depth While transforming the original formula into negation normal form, TEMPLAR evaluates the maximum formula depth of the original formula. This metric, together with the clause count may be usable for the decision of the theorem prover which clause form (conjunctive/disjunctive or definitional) should be chosen.

Structure Containment Some theorem provers have specialized inference engines which handle special sets of formulae. For example, a theorem prover could use specialized heuristics when it is aware that equalities are existent or absent in the given formula set. The same holds for the case that single-quoted strings or numbers are contained. Thus, TEMPLAR determines whether equalities, single-quoted strings and numbers are existent in a formula and may output this information to the theorem prover. This informations are generated during the negation normal form transformation.

Equality Scope We will now consider classical first order logic with equalities for which there is no optimal solution in automated theorem proving. If two predicates are not unifiable due to some function symbols being incompatible, there may still be an equality in the source or the target formula which states that those two functions are equal. After the negation normal form transformation, the equality scopes for a formula are created iff the formula contains equalities. The equality scope is a virtual relation that states for one predicate (or equality) and a set of equalities that if the formula would be transformed into disjunctive normal form, the predicate (or equality) would occur together with every equality in the set in at least one of the clauses. In conjunctive normal form in contrast, the predicate would not be in the same clause as the scoping equalities. This scope is created by using the laws of distributivity. Despite extensive search, no literature was found which describes such an analysis with this scope so that it is assumed that this technique is either new or not well-known.

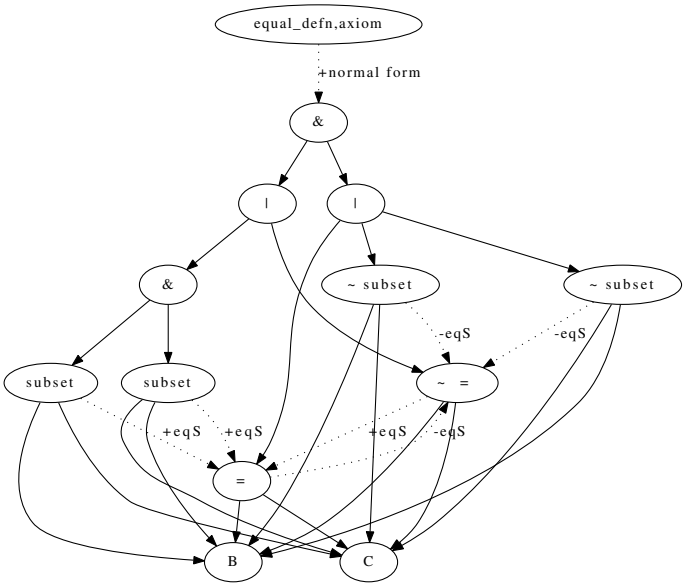

Figure 3.2: Formula in negation normal form with visualized equality scope

Figure 3.2 shows the formula *equal_defn*, again with visualised equality scope. The dotted relations show that the target of the relation scopes the source of the relation. A relation *+eqS* states that the scope is one of a positive equality and *-eqS* is a scope of a negative equality.

The equality scope of one predicate can be used by TEMPLAR to apply unification via equalities and has the aim to reduce the number of selectable equalities for the E-Unification.

3.3 Unification based Search Strategies

This section covers the search for connections in the set of all formulae. In especial, different search strategies with different properties are introduced and described in depth. First of all, the basics of the different selection algorithms are described, followed by a description of search graph restrictions for specific algorithms.

3.3.1 Selection Algorithm Basics

This part illustrates the basic concepts of all selection algorithms. Here, the selection process which is used by the unification based selection strategies is described. Based on this, the basic selection, the non-structural and the structural selection will be introduced.

Selection Process Every unification based selection makes a breadth first search starting with a seed. The selection of the seed is not trivial since not every formula is adequate for initializing the selection. For example, the conjecture could be a formula like **$false** which states that the formula is false. This is commonly known as the liar's paradox [Rus08]. If the formula does not contain any predicate which can be connected, we need to use another formula as seed. The TPTP specifies several formula roles including conjecture/question, definition, lemma, theorem, hypothesis, assumption and axiom. According to the TPTP technical report, the formulae with the types lemma and theorem are derived from the axioms and thus must be provable from the axioms. If the conjecture cannot be used as seed, other formula types may be used.

Thus, a list of seeds is generated with the following precedence:

1. conjectures/questions
2. theorems
3. lemmas
4. hypothesis
5. assumptions

If none of the conjectures can be used to create a connection graph, try to create a connection graph for a theorem. If this fails, try to use the lemmas as seeds and so on. Axioms are not used as seeds since this would require to chose an axiom randomly which is not feasible when having many thousands of axioms.

When the initial search for connection is successful (first iteration), connections are searched for the now included (connected) formulae (second iteration). After that, connections are always

searched for the formulae which were connected in the last iteration. The process halts if a proof was found or the time expired.

After every iteration, a prover instance is started if wished.

Basic Selection This selection is based on the concept of ARDE which was described in [Fra12b]. Every formula is seen as a set of predicates which is a relatively weak model but easy to implement.

In the initial iteration, a connection is searched for every predicate contained in the seed formula. Since every connection has a predicate as target and the target is contained in a formula, the target formula can be seen as connected to the source formula and is included. In every following iteration, connections are searched for formulae which were connected in the last iteration for the first time. If a formula is connected in the second iteration for example, every possible connections to the predicates of this formula are ignored (no predicate of an already included formula is used as target of a connection) in the third and every following iteration since the formula was already included. Also, every variable substitution which is a consequence of an unification process is discarded.

This search model is obviously not a graph search in it's common way. It is a layer based search algorithm and all iteration results are pairwise disjunct sets of formulae.

To be precise, let $i, j >= 0$ be iteration indexes and $Selected_i$ be the set of formulae which were selected in iteration i (or j). Then,

$\forall i \geq 0. \forall 0 < j < i.\ Selected_i \cap Selected_j = \emptyset$ must be true.

Non-Structural Selection This selection is a refinement of the basic selection and uses the variable substitutions which lead to the unifiability of two predicates. The aim is to reduce the possible connections in later iterations by binding the variables with the unifier. This involves the need to break with the layer-based representation of the search. In a layer, there are no information which two predicates were connected and thus lead to the inclusion of a formula into the selected formula set. But if we want to use the variable substitutions, we need chains of connections which contain the variable substitutions, the source predicate and the target predicate.

This implies a graph-representation of the search. If a predicate was the target of a connection from multiple different predicates, potentially contained in different formulae with potentially different variable substitutions, we need to consider the connections as separate result sets.

The initial iteration is conceptionally equivalent to the one for the basic selection. But in contrast to the basic selection, including a formula multiple times is possible. For every predicate in the seed formula, connections are searched and saved as connection chain with initial length of one. The variable substitutions applied to the source and target predicate are saved in the connection chain. In every following iteration, every chain is tried to be extended by connecting all predicates in the target formula which were not yet connected in the chain.

Structural Selection The Structural selection is a strong refinement to the non-structural selection. In contrast to the non-structural selection concept, the structure of a formula for which connections are searched is considered. Since the search process is refutation-driven, the formula structure is extremely valuable.

Consider an arbitrary formula of first order logic which shall be refuted. To achieve this, we need to find a set of connections from the predicates of this formula to predicates of another formula - but not necessarily a different one. In especial, consider the two different types of compound formulae in conjunctive normal form - conjunction and disjunction. A conjunction is refutable iff there is at least one contained predicate which is refutable and a disjunction is refutable iff for all contained predicates a refutation exists. If we can not refute all predicates in a disjunction, the entire disjunction cannot be refutable and we can discard at least all found connections for this disjunction. If we can not refute even one predicate of a conjunction, the entire conjunction can not be refutable.

These rules are used as restriction. Since the search does not use the clause normal form (conjunctive or disjunctive) but the negation normal form, nested formulae can lead to even more restrictions. Consider a disjunction which contains at least one conjunction and some predicates. If the conjunction is not refutable, all connections of the disjunctions are discarded. But if one of the connections in the disjunction or conjunction was the one which lead to the inclusion of the formula, we can roll back the inclusion - the including connection becomes invalid.

But for this, we need a slightly different representation of the search graph. every connection needs to know which two predicates and which variable substitution were used but also which outgoing connections were established and which connection lead to the construction of it. Every connection needs to know it's source connection.

3.3.2 Search Graph Restrictions

In this part, the restrictions to the search graph for the non-structural and structural selection are described.

Spanning a search graph can lead to an enormous growth of the search space. This is due to the fact that the use of the search graph leads to the possibility of visiting a predicate more than once. But this is needed since a predicate normally can be used for different connections with different variable bindings. And the use of different variable bindings is explicitly wished since it can reduce the number of possible connections when other constraints apply like the use of the formula structure.

Thus, pruning techniques are needed to reduce the search space. Those are the Reconnection Suppression, the Ground Connection Target Merging, the Conjunction Cut and the Nexus Path Union.

Reconnection Suppression This technique is quite simple and should be intuitive. Since the search is represented by paths and in every path, the visited predicates are known, the connection of already visited predicates should be omitted. This can be done since re-connecting will not change the search space in any way apart from possibly creating circular loops which should obviously be avoided.

Ground Connection Target Merging This technique uses the identity of multiple target predicates to prune the search space. Consider the two formulae F1 and F2 as shown in figure 3.3. While F1 is a fact, F2 contains three predicates of which two are identical. As this figure shows, two connections are possible and are normally evaluated. Note that both connections are ground

which means that no variables are involved. In both branches, two predicates need to be connected which leads to at least four connections when the non-structural search is used.

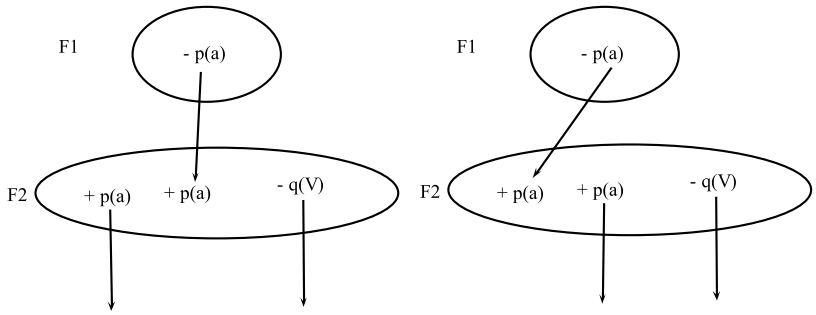

Figure 3.3: Two connections using syntactically identical targets

But since both connections between F1 and F2 are identical, they can be merged into one connection, marking both target predicates as visited. The application of this pruning leads to a connection depicted in figure 3.4.

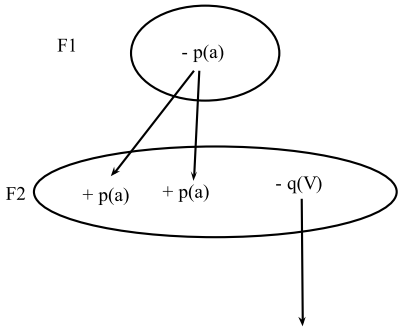

Figure 3.4: Two merged connections using syntactically identical targets

This merging omits the branch and leads to the reduction of four predicates to one predicate which has to be revised for connection.

This pruning technique can be applied for non-structural and structural graph search.

Conjunction Cut The conjunction cut is a set of pruning techniques which are used for the structural search since the formula structure is needed. The first one is a technique for choosing connections to follow and uses the fact that one connection into a conjunction is sufficient to refute it and uses ground connections. Consider the formulae shown in figure 3.5 where F2 is a conjunctive formula.

All of the shown connections to F2 are valid refutations. Thus, choosing one of the connections is sufficient and all others can be discarded. But if F2 is just a sub formula of a disjunction, choosing the one connection which binds a variable could be the wrong choice when the variable binding is not the right one for a proof. So, taking a ground connection should be the most secure way since this results in minimal restrictions to the further search graph. This leads to a quite simple rule. Whenever there are multiple connections from one predicate to multiple predicates in

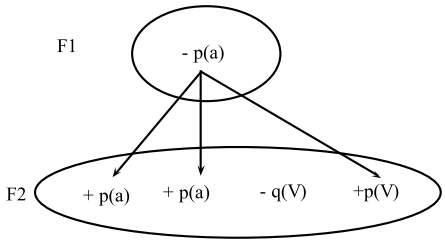

Figure 3.5: Three connections to one conjunctive formula

one conjunctive formula, choose a ground connection, if existent. If there is no ground connection, choose all connections.

The second technique reduces the count of possible outgoing connections depending on the ingoing connection to the formula. It tries to reduce the drawbacks of not having a clause form. One of the drawbacks is the nested structure of the formulae. Consider a connection which leads into a conjunctive formula when having a clause form. This connection eliminates the complete clause. But when the formula is not in clause form but in negation normal form and the predicate is directly contained in a disjunction, the disjunction cannot be eliminated.

To see this problem and the solution, consider the formula $(\neg p(x) \wedge q(x)) \vee (r(x))$, given in figure 3.6 in a matrix representation.

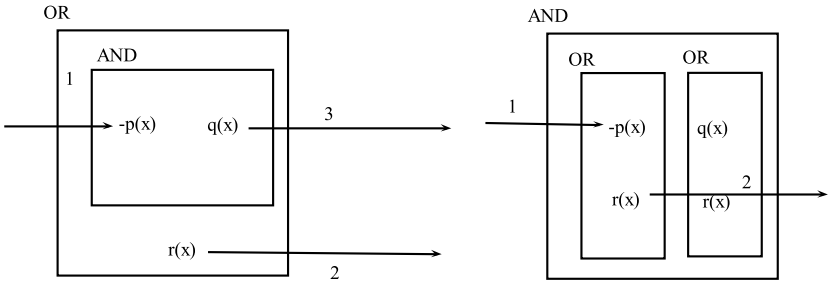

Figure 3.6: The matrix representation of the formula $(\neg p(x) \wedge q(x)) \vee (r(x))$ in negation- and conjunctive normal form

In conjunctive normal form, it would be clear that the connection 3 can be omitted. But in negation normal form, this is not clear when having deeply nested matrices. But the predicates $\neg p(x)$ and $q(x)$ have a common super formula which is a conjunction. In especial, the conjunction is the most inner common super formula and this fact can be used to simulate the knowledge existent in the clause normal form. If the law of distributivity would be applied, the two predicates would not be in the same clause.

This leads to a rule for omitting outgoing connections depending on the ingoing connection. If the most inner super formula of the predicate to connect and the predicate which was used to visit the formula is a conjunction, do not connect. But the complexity of this check should not be forgotten. Consider two predicates of which one lies in depth i and the other in depth j where $i, j > 1$. If the common super formula has to be evaluated and one starts with the predicate in depth i, one needs to check i times for all j whether it is the same super formula and whether it is a conjunction. But when having the information for both predicates, which super formulae contain

those predicates, this is possible in linear time. If every predicate knows the set of conjunctive formulae which contain it recursively, the containment of a formula in this set can be checked in constant time. Then, only n checks are needed where n is the count of conjunctions which contain the predicate which is considered. But what if a formula is included by m distinct paths? In every path, and for every predicate which could be connectible, this check would be done. This, still is not efficient enough. But to reduce the count of checks, more information about the formulae are needed. The solution is to compute a closure for every predicate (and equality) which contains references to all predicates whose nearest common super formula with the specific predicate is a conjunction. In a matrix representation as it is used to describe the connection calculus, all predicates which have a conjunction as nearest common super formula are next to each other in horizontal direction. All predicates which share a disjunction are next to each other in vertical direction. So, if a connection is found into a formula, all predicates which are in the closure of the target predicate of the connection can be marked as irrelevant and will not be used for outgoing connection. Since the conjunctive closure is an extension of the equality closure, it can also be created almost completely during the negation normal form transformation.

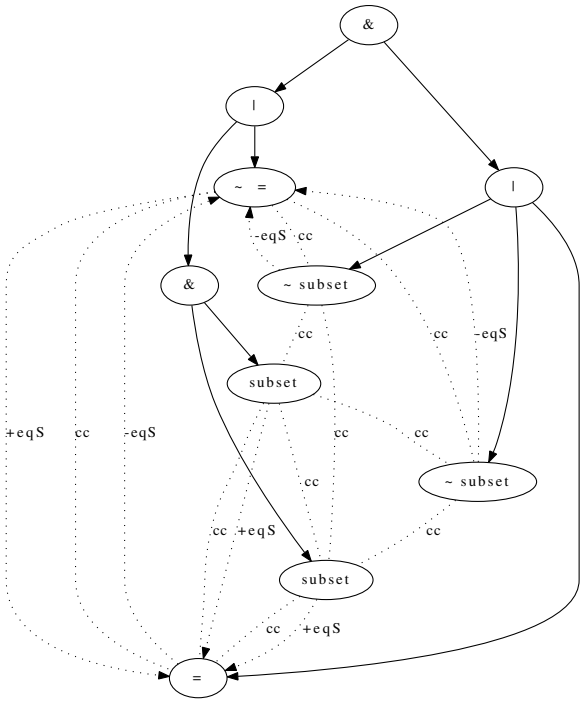

Figure 3.7: Formula in negation normal form with visualized equality scope and conjunctive closure

Figure 3.7 shows the formula *equal_defn*, again but this time with equality scopes and the conjunctive closures which are marked with the dotted relation *cc*. The variables are removed in order to make the figure better printable. The semantics of this relation are the same as those of the equality scope relation. To see that the the conjunction closure is an extension of the equality scope, it should be sufficient to see that for every equality scope relation, there is one conjunctive closure relation with the same source and the same target but the conjunctive closure relation is an undirected one.

Nexus Path Union The Nexus Path Union is the only technique in TEMPLAR which breaks the independence of distinct paths. This independence is quite important since different paths normally contain different variable substitutions which should not be mixed up. But in order to reduce the search space, the union of paths should be done, if appropriate. Consider two paths, both connecting into the same formula for the first time and thus including it. If both connections target the same predicate and the connection is ground, the extensions of both paths must be identical since the variable state of the target formula is identical. But what if the extension fails and the path becomes invalid? Both paths would have to be rolled back. And if they do not become invalid, they are completely redundant. This is a spot where pruning can be applied. If one path connects a predicate in a ground manner and the target formula is visited for the first time, the incident may be saved. If another path ground-connects to the same predicate and also for the first time into the formula, this path can be marked as alternative source connection path for all extending paths of the first path which ground-connected into the predicate. If the first path happens to be invalid, the connection into the predicate can be discarded, possibly yielding an invalidity of the second path.

Surely, this technique can only be applied to the structural search since invalid paths can only occur if the formula structure is used.

In the next section, the frequency based selection is conceptually described.

3.4 Frequency Based Selection

This section describes a more linguistic approach on the selection of relevant formulae which has some common parts to the SinE selection.

The selection uses a tf-idf weight which is applied to every lexeme (function classifier, constant, equality and predicate) and is created in the following steps.

1. calculate the raw frequency of all lexeme in the formula
2. calculate the normalised text frequency based on the raw frequency.

While the first step merely counts the number of occurrences of each lexeme in the formula, the second step uses a normalisation function like the division of the raw frequency by the raw frequency of the most common lexeme in the formula.

After this steps are done for all given formulae, the inverse document frequency is calculated for all lexemes. First, the count of formulae in which the lexeme is contained is calculated (raw document frequency). After that, the count of formulae is divided by the raw document frequency and the logarithm of the result is taken.

But the lexemes with the highest tf-idf value are not necessarily the most important. Thus, the relevance of lexemes is based on a range of th-idf values. A lexeme is seen as relevant for a formula if it's tf-idf weight is at most n times lower than the most relevant lexeme (comparable to the "triggers" relation in SinE).

With this triggers, the selection is done. The conjecture is included as relevant in the first iteration of the selection process. In every iteration after that one, the following steps are applied.

For every formula which was included in the last iteration:

1. if the maximum iteration was reached, end the search

2. get the lexemes which trigger this formula
3. search for all formulae which are triggered by those lexemes, too
4. include all formulae which are triggered and not already included

This search can lead to the inclusion of (too) many (not relevant) formulae which is not the wished result. Thus, a finer granularity is needed. Consider a conjecture which has more than one triggering lexeme. It is possible that not all lexemes lead to the needed formulae in the search but only a subset. In this case, it is more feasible to perform the search per relevant lexeme (seed) of the conjecture. But it is also possible that multiple seeds are needed in combination. Thus, the search is performed for every seed and after that, the formula sets of all seeds can be merged per iteration.

4. Implementation

This chapter describes the implementation of TEMPLAR which includes the import and representation of the data, normal form transformations, analysis of the formulae, the search algorithms and the output. First, an overview of the workflow of TEMPLAR is given in order to make the modularity clear.

4.1 Overall Workflow

The workflow of TEMPLAR is managed by three parts which work hand in hand. The top-level of the workflow is the TEMPLAR main function which triggers the subsystems. The overall workflow is shown in figure 4.1 which is a Unified Modelling Language (UML)[1] sequence diagram.

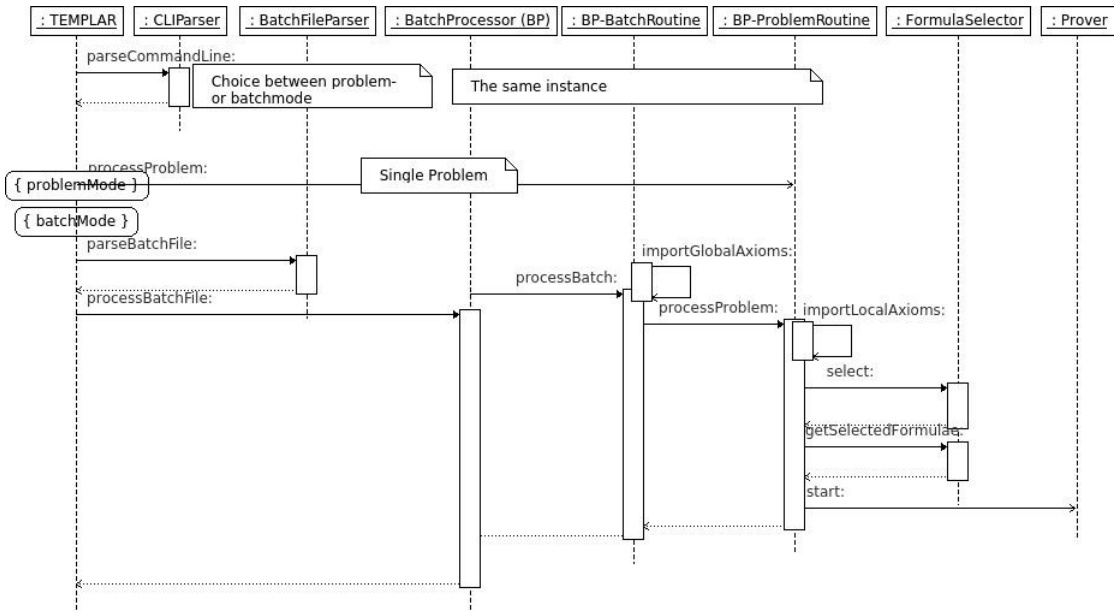

Figure 4.1: The overall workflow of TEMPLAR

As mentioned, TEMPLAR is the top-level management layer. It's first action is to trigger the command line parsing. If this one is successful, a configuration object called TEMPLARConfiguration is generated which contains the following information:

- The input file name (mandatory)
- Whether the input file is a batch file or not (default: no batch file but TPTP file)
- Whether proofs should be attempted (default: no attempts)
- The chosen search engine (basic/non-structural/structural/frequency based/dynamic selection, default: dynamic)
- The output form of the formulae (original or Skolem, default: original)

[1] http://uml.org

- Whether conjectures should be seen as valid and axioms as invalid, default is invalid conjectures
- The global wall clock time (default: unlimited)
- The CPU core count (default: 4)

If the batch mode is activated, a batch file parser is triggered which parses the file and returns a vector of Batch objects which contain informations like global axiom files, problem time limits and the problem input and output files. In batch mode, the BatchProcessor is triggered with the TEMPLARConfiguration and the vector of Batch objects. The BatchProcessor itself iterates over the vector of Batch objects. In every iteration, global axiom files are imported (concurrently, if there are enough cores and files). The global axiom files are held persistent until the complete batch is finished in order to reduce the count of files which need to be loaded again. After that, the BatchProcessor iterates over all problem definitions in the Batch object and imports the problem file and all local axiom files. The local axiom files are cached for later problems and batches. If a local axiom file has the same path as a global one, the duplicate is recognised. After loading all files, the formula selection is triggered. This is done by creating a FormulaSelector object which is one of the search engines. Then, a chain of functions is called until no new formulae are selected or the time limit is reached (which is checked by TEMPLAR). After every iteration (*select*,*getSelectedFormulae* and check whether new formulae were selected), a prover instance is started on the selected formulae if proofs should be attempted. If one of the prover instances finds a proof before the time limit is reached, all other prover instances and the selection are stopped.

If the batch mode is not activated, the input file is seen as TPTP file and a direct call to the problem routine is done. For this mode, the ProblemProcessor was implemented. It only imports the problem file and local axiom files, if existent and uses the FormulaSelector and the prover in the same way the BatchProcessor does.

Now that the general workflow is clear, the specific parts can be described in-depth.

4.2 Data Import and Representation

This section concerns the import and representation of the data. The first order formulae need to be imported from the TPTP syntax into a structure which can be processed by the system.

4.2.1 Representation of the Formulae

In order to process the formulae, specific parts of them need to be distinguishable from each other. For example, a variable needs to be distinguishable from a constant, function, number or quoted string. Thus, all variants of terms need to have specific representations. But since numbers and quoted strings are constants, they have some common properties with constants. Though constants formally are functions with an arity of zero, they are not seen as a subtype in this implementation. The reason is that checking whether two terms are equal is easier to implement when constants are not treated as functions. If the constant would be treated as function, at least the arity comparison would be needed which is a waste of time when it is possible just to check whether both terms are constants. Furthermore, no argument vector needs to be saved for constants which would be empty anyway but still use a minimum amount of memory.

All this implies the implemented class hierarchy for terms as shown in figure 4.2 which is an UML class diagram. All UML class diagrams used in the following may contain the following relations:

- Generalization (arrow with triangle tip): the target is the generalization of the source
- Aggregation (arrow with non-filled diamond): the target has an object of the source type
- Composition (arrow with filled diamond): the target contains the source, if the target is destroyed then the source is, too.
- Association (dotted arrow): the target is somehow associated with the source, e.g by aggregation

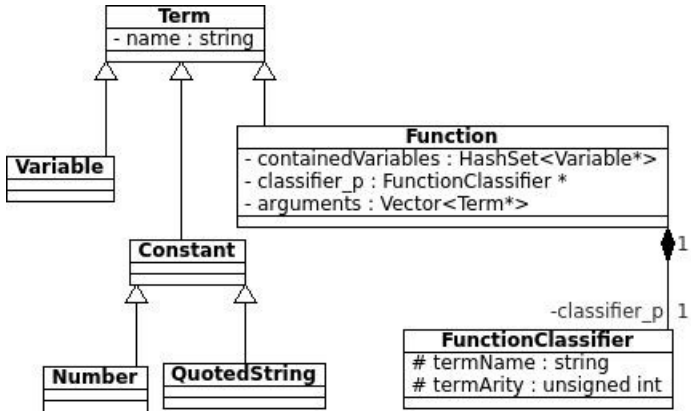

Figure 4.2: The Hierarchy of Terms

Though the figure is self-explanatory, there are some details which are important. Some of the attributes of the classes contain the symbol * which denotes that the attribute is a pointer or more specific, a variable which saves the unique memory address of the object. This already gives a small insight to the way the unification works. The comparisons of constants, variables and functions are not done by comparing the name (and arity for function) but by merely comparing the fixed-size memory addresses. If a constant is used in two functions, the memory address is the same for both occurrences. For functions, the memory address cannot be used since a function with a specific arity and name can contain different arguments in different occurrences. Thus, the Function class is merely a container for the argument vector and an object which is called FunctionClassifier. The FunctionClassifier encapsulates the name and arity which allows the unification of functions with the same classifier but different argument vectors. Obviously, the check for the equality of name and arity can be done in one step by merely comparing the memory address of the classifier objects. Also, the Function class contains a HashSet of Variable pointers which is the set of all variables which are contained in the function. This allows an occurs-check with constant time complexity in the average case for the unification.

Since the formula structure is used by TEMPLAR, the formula structure needs to be represented in a usable way, too. The figure 4.3 shows the class hierarchy for formulae which is quite complex due to the complexity of the TPTP syntax.

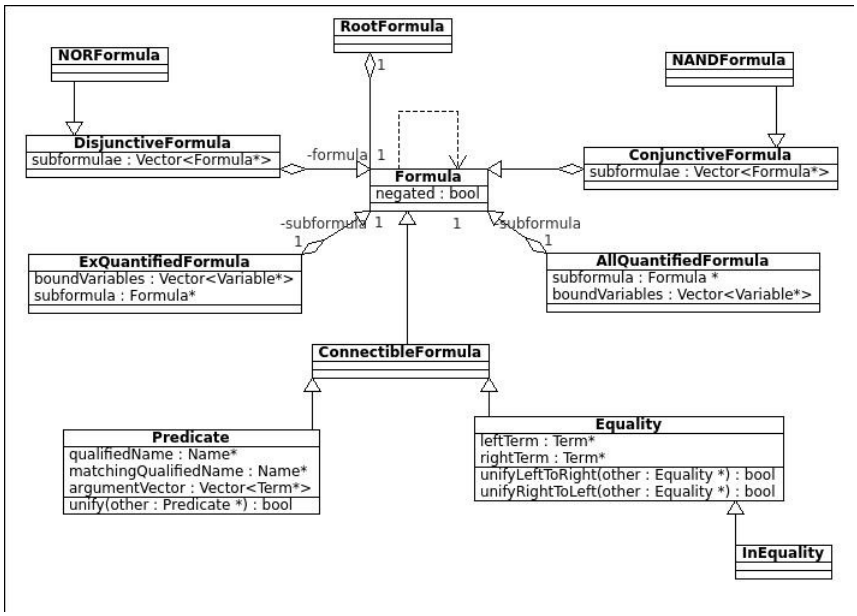

Figure 4.3: The Hierarchy of Formulae

In order to make the processing of the formulae as easy as possible, some inheritances were introduced. The negated equality ($\sim=$) inherits from the equality and just behaves inverse to the equality but has the same properties. The negated disjunctions and conjunctions inherit from the disjunction and conjunction though the TPTP specifies NAND and NOR as non-associative operators while disjunction and conjunction are specified to be associative. The reason is that the negation normal form transformations can be implemented in the DisjunctiveFormula and ConjunctiveFormula class and the NORFormula and NANDFormula class just calls the inverse transformation functions in their base classes. The Equality class has two unification functions since equalities are symmetric and thus, two unifications are needed to check whether two equalities are unifiable. The Predicate class contains two Name object (string container) pointer which are the qualified name of the predicate and the qualified name of the predicate class which is a candidate for connection. The reason for using the Name class is that predicate classes are quite common and may become long. Thus, saving the full name once and using an fixed-size pointer can spare a lot of memory if the average length of the predicate class name is higher than eight. This is due to the fact that a pointer uses eight byte on a 64 bit architecture which is equivalent to a string of size eight and the predicate class consists of the polarity (one byte), the predicate name and the arity (at least two bytes since a slash is used as delimiter). Thus, an average predicate name size of six or higher leads to reduction of memory consumption. Also, the predicate classes can be held persistent in a hash table which has pointers instead of possibly long strings as key and sets of Predicate pointers as value. The access can be expected to be faster with pointers since memory addresses are unique which allows some optimizations for the hashing algorithm of hash tables. In order to save equalities as (Equality)predicate classes, too an abstraction is needed. This is the purpose of the ConnectibleFormula class. It is a generalization for all formulae which can be used for a connection process.

4.2.2 Import of Formulae with Structure Sharing

As it was stated in the concept chapter, structure sharing is a crucial part of TEMPLAR. There are two possible ways to assure structure sharing. The first one is to import all data and then apply structure sharing. But since this leads to higher memory consumption and takes much computation time, a second approach is used. This approach applies structure sharing already during the parsing of the TPTP files. But for this, a central container is needed which holds all structures persistent and can deliver every structure which is existent and also can construct them. This is quite easy to implement when only one file has to be parsed at a time. But when the parsing of multiple files shall be concurrent, there are some problems which need to be addressed. First of all, the container needs to synchronize access to the structures. And every parser which processes one file needs to be able to construct complex formulae or non-constant terms which are not shared.

Structure Sharing The way the structure sharing is assured already during the parsing step in TEMPLAR is depicted in figure 4.4.

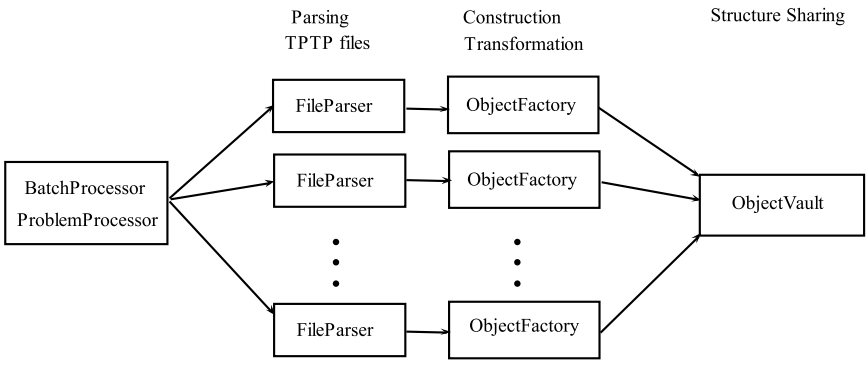

Figure 4.4: Assertion of Structure Sharing with concurrent Parsing

The ObjectVault is the container which holds all primitive structures like constants, numbers, quoted strings and Skolem id counters persistent and also can create them. It implements the Singleton Pattern (described in [GHJV95], for example) which assures that there can only be one instance of the class during the complete lifetime of TEMPLAR. The synchronization of access is assured by mutual exclusion (mutex)[Tan07] constructs which are locked on access and unlocked when the access is not needed any more. The module which processes a problem or a complete batch file can start multiple TPTP file parsing functions concurrently and assign an ObjectFactory to the specific parsing thread. All factories are independent to each other and each of them gets construction queries from the parser they were assigned to. The factories produce formulae or terms with higher complexity like functions, predicates, disjunctions and so on. If a constant, for example, is requested, the factories query the ObjectVault for the construction. Whenever a TPTP formula was successfully parsed, the assigned ObjectFactory constructs the Skolem normal form for the formula, creates the conjunctive closure and equality closure. More precise, the ObjectFactory calls the RootFormula (the structure which encapsulates the logic formula) with a method to transform itself into normal form.

The number of parsing threads is limited since it does not make sense to create six threads on a dual core processor. Also, it would not make much sense to construct four parsing threads when

there are only two files to parse. Thus, the limit of threads ($count_{threads}$) is calculated with the formula shown in figure 4.5.

$$count_{threads}(count_{files}, count_{cores}) = \begin{cases} count_{files} & count_{files} \leq (count_{cores} - 1) \\ count_{cores} - 1 & count_{files} > (count_{cores} - 1) \end{cases}$$

Figure 4.5: Formula for calculating the parsing thread count

The reason for using not all cores for parsing is that the operating system usually needs a core, too since file operations like reading are not directly done by TEMPLAR but by the operating system. The files to parse are just enqueued in a synchronised First In First Out (FIFO)[Tan07] queue whose memory address is given to all threads which take files from the queue until it is empty.

> **Note:** Structure sharing has some impact to the memory management. Since constants, for example are shared, and thus are not allowed to be removed from memory when one formula is deleted which uses them, a possibility is needed to keep the data consistent. In TEMPLAR, every shared structure has a multiplicity value which is the number of references that are held to them and there is no direct way to delete them (which is done by a destructor in C++). The destructor is hidden from external access in order to prevent the structures from being deleted. If a shared structure needs to be used, e.g. for unification and would be copied, a function called *getInstance* needs to be called which increases the multiplicity by one. If a reference to a shared structure is not needed any more, the function *release* has to be called which decrements the multiplicity. The access to these functions is synchronized by mutexes in order to make them thread-safe. If the multiplicity of a shared structure is decreased to zero, the structure de-registers itself from the ObjectVault and deletes itself.

TPTP Parsing Grammar The parsing is done by a grammar which constructs a formula graph for every formula by applying semantic actions which include the access to the ObjectFactory. The grammar is implemented in ordinary C++ syntax but in a more declarative manner which omits the specific implementation of the parsing. It has some similarities with the Extended Backus-Naur Form (EBNF) and is transformed into a real parser by the boost spirit[2] parser generator. Though the TPTP syntax definition is precise and good for checking grammars, it is not really tailored for grammars which generate the parsed formulae. One of the reasons can be seen in the listing 4.1.

```
1 <fof_logic_formula>  ::= <fof_binary_formula> | <
     fof_unitary_formula>
2 <fof_binary_formula> ::= <fof_binary_nonassoc> | <
     fof_binary_assoc>
3 <fof_binary_nonassoc> ::= <fof_unitary_formula> <
     binary_connective> <fof_unitary_formula>
4 <fof_binary_assoc>   ::= <fof_or_formula> | <fof_and_formula>
5 <fof_unitary_formula> ::= <fof_quantified_formula> | <
     fof_unary_formula> |
```

[2] http://boost-spirit.com

```
6                                   <atomic_formula> | (<fof_logic_formula
                                       >)
```

Listing 4.1: Excerpt of the TPTP fof syntax definition for logic formulae

The formula (*fof_logic_formula*) can be a binary formula or an unitary (predicate, negated formula, quantified formula and so on). A binary formula could be an associative formula (& or |) or a non-associative ($\sim$ & or $\sim$ |). So consider a formula which is just a predicate. First, the rule for *fof_binary_formula* would be chosen which first leads to the rule for *fof_binary_nonassoc* which parses two unitary formulae with for example NAND as junction. This rule would fail but would have produced the predicate in-memory since a predicate is an unitary formula. Next, the parser would try to apply the rule for associative binary formulae and would fail both in *fof_or_formula* and *fof_and_formula* and produce the predicate in-memory again. Then, the parser would find the rule for unitary formulae where he would finally find the matching rule. So, there would be at least four wasted constructions of the predicate.

But the grammar can be adapted for a generating purpose. This is done in TEMPLAR and looks like the C++ code in the listing 4.2.

```
1 formula = unitaryFormula[_a=_1] >>
2         (/// Non-Associative
3         (binaryConnective >> unitaryFormula)
4         | /// Associative
5         conjunctiveVector | disjunctiveVector
6         |/// only unitaryFormula
7         eps[_val=_a]
8         );
```

Listing 4.2: Disambiguated definition for TPTP formulae

Since an unitary formula is part of every rule in the syntax definition, it has to be in first position and the result of the rule for unitaryFormula (*_1*) can be saved in a variable (*_a*). After that, the following formula parts may follow in this order:

1. $\sim$ & *unitaryForula* or $\sim$ | *unitaryForula*
2. (& *unitaryFormula*)$^+$ or (| *unitaryFormula*)$^+$
3. the empty word which results in the unitary formula being the result (*_val*)

In all cases, no additional objects are created since the connectives lead to a failing sub-rule and the already saved unitary formula is not lost.

The other source of ambiguities is the fact that equalities are used infix ($X = Y$). The listing 4.3 shows another excerpt which contains the definitions for predicates, infix equalities and infix inequalities.

```
1 <fof_unitary_formula> ::= <fof_quantified_formula> |
2     <fof_unary_formula> | <atomic_formula> |
3     (<fof_logic_formula>)
4 <fof_unary_formula>  ::= <unary_connective>
```

```
5       <fof_unitary_formula> | <fol_infix_unary>
6   <fol_infix_unary>     ::= <term> <infix_inequality> <term>
7   <atomic_formula>      ::= <plain_atomic_formula> |
8       <defined_atomic_formula> | <system_atomic_formula>
9   <plain_atomic_formula> ::= <plain_term>
10  <plain_atomic_formula> :== <proposition> |
11                             <predicate>(<arguments>)
12  <defined_atomic_formula> ::= <defined_plain_formula> |
13                               <defined_infix_formula>
14  <defined_infix_formula>  ::= <term> <defined_infix_pred>
15                               <term>
16  <defined_infix_pred> ::= <infix_equality>
```

Listing 4.3: Excerpt of the TPTP fof syntax definition for unitary formulae

An infix inequality is a *fol_infix_unary* which is a *fof_unary_formula* while the infix equality is a *defined_infix_formula* and also a *defined_atomic_formula*. The predicate is a *plain_atomic_formula* which is, as the infix equality an *atomic_formula*. This segmentations lead to an ambiguity which needs to be resolved.

Consider the formula $p(a,b,c) = q$ which obviously is an equality. Following the precedence in the rule *fof_unitary_formula*, the sub-rule unary formula would be chosen since quantified formulae have a quantifier and the rule for them would not match. In the rule *fof_unary_formula*, the fol infix unary branch would be taken which leads to the creation of the term $p(a,b,c)$ which is correct. But since the formula is no inequality, the rule would fail and the term would be lost. After that, the rule *atomic_formula* and in that one, the sub-rule *plain_atomic_formula* and in the end, a predicate $p(a,b,c)$ would be constructed and after failure, lost again. This shows the overall problem. No matter how the precedences in *fof_unitary_formula* are changed between the unary and atomic formula branch. Either there will be false creations of terms or of predicates. Thus, another grammar is needed for this purpose. The grammar which is used in TEMPLAR is shown in listing 4.4.

```
1  equalityOrPredicate =
2        // equality since only vars are upper-case
3      ( variable >> equalityType >> term )
4      | // or
5      (
6        (
7            // save name and arguments
8            // still could be term or predicate
9          ( lowerCaseWord >> (arguments | eps ) )
10           // or some kind of constant
11         | quotedString | complexNumber
12       ) >> // followed by
13       (    // either "= term" or "!= term"
14            // -> construct equality
15         ( equalityType >> term )
```

```
16          | // or nothing -> construct predicate
17          eps
18        )
19    );
```

Listing 4.4: Disambiguated definition for TPTP predicates, equalities and inequalities

If the input to parse starts with an upper case letter, the first part can only be a variable and thus, the complete input has to be an equality or inequality if it is conform to the syntax. If the input to parse starts lower-case, the first part may be a term or predicate. Thus, the name and, if existent, the arguments are saved. From this saved information, both an equality and a predicate can be constructed. With this grammar, equalities, inequalities and predicates can be parsed unambiguously.

In the following section, the implementation of the normal form transformations is described. Also, the analysis of the formulae, in especial the generation of the metrics like formula depth and clause count is described.

4.3 Normal Form Transformations and Analysis

To use the results of the analysis and transformation, a way is needed to save the results. This is the purpose of the class RootFormula. But since there are different types of formulae, this abstraction is to high. Thus, the RootFormula is just the base class of the Axiom class and Conjecture class. The RootFormula class has the following attributes which are partly TPTP-specific and partly TEMPLAR-specific.

- Formula name
- Formula role (conjecture, axiom, lemma, etc.)
- List of Predicates (used for the basic selection)
- Original formula graph (Formula* created by the import and used for output)
- Normal form graph (Formula*, created by the normal form transformation, used for search)
- Formula depth (integer) of the normal form (may be lower than the original formula depth)
- Equality containment (boolean value, used for deciding whether the equality scope should be calculated)
- Number and QuotedString containment (boolean value)
- Conjunctive Normal Form (CNF) clause count (integer, worst-case value)

Some of the attributes of the RootFormula objects are created during the parsing. These are the formula name, role and the original formula graph. The other attributes need to be set after that. Obviously, the normal form graph is created during the normal form transformation. But the transformation, as it is implemented does much more. In this step, the list of predicates, the

formula depth, the Equality, Number and QuotedString containment and also the clause count are determined. Some of the determined attributes can be generated trivially. For example, the Equality containment and also the Number and QuotedString containment is simply set by giving the boolean attributes as pointers to the transformation function and setting the value to true when an Equality (or InEquality) or Numbers and QuotedStrings are visited, respectively.

But the normal form transformation does much more. Whenever a TPTP formula was successfully parsed, the ObjectFactory constructs an Axiom or Conjecture object, depending on the role of the TPTP formula. All formulae which are not conjectures are constructed as Axioms since they are defined to be true by the TPTP definition. After constructing the object, the ObjectFactory sends a request to the object to transform itself into normal form and gives some parameters to the transformation function which include the following but are not limited to them.

- Pointer to the Equality containment variable
- Pointer to the Number/QuotedString containment variable
- Pointer to the formula depth variable
- Pointer to the clause count variable

The transformation itself is done by the RootFormula. If the current RootFormula is an Axiom, it transforms itself with positive polarity. If it is a Conjecture, it transforms itself with negative polarity. This behaviour can be inverted by a command line option. The RootFormula itself sends a request to the encapsulated Formula object to transform itself into normal form and return the normal form formula. This request can be a positive or negative transformation, depending on the subtype of the RootFormula. The Pointers are handed down to the top-level Formula object and an initial formula depth of zero is given as parameter, too. The Formula, recursively, hands down the parameters to it's sub-formulae. Whenever a sub-formula gets the transformation request, it hands down the current depth incremented by one. If the current formula is a Predicate, an Equality of InEquality, the current depth is compared to the depth which is saved in the RootFormula (*maxdepth*) object. If the current depth is higher than the value in the RootFormula, the *maxdepth* is set to the current one. Whenever an Equality is visited, the respective variable is set to true. The same holds for Number objects and QuotedString objects. The clause count is calculated, too. Whenever a Formula would transform itself into a disjunction (e.g. negative transformation of an Equivalence), the clause counts of the sub-formulae would be multiplied. Whenever a Formula would construct itself into a conjunction (e.g. negative transformation of an implication), the clause counts of the sub-formulae would be added. Whenever a Predicate, Equality or InEquality object is transformed, it returns a clause count of one for itself to a parameter which was passed to it. This complies to the clause count calculation formulae in paragraph 3.2 of section 3. In order to make the normal form as small as possible, flattening is applied. If a Disjunction is constructed, for example and the constructed normal form of a part of the Disjunction contains a Disjunction, the sub-formulae of the embedded Disjunction are lifted to a higher level.

Note: The normal form depth does currently not consider the flattening. The reason is that there is no a priori knowledge about the formula type of the transformed sub-formulae. Thus, there is no way to calculate the flattened formula depth during the normal form transformation.

The skolemization is also done during the normal form transformation in order to reduce the number of graph traversions. Since the Formula objects also get a vector (precisely, the copy of the vector which was passed to the super formula) of the variables which were bound universally in the outer scope, this is also passed to the quantified formulae.

Also, a HashTable (precisely, the copy of the HashTable which was passed to the super formula) is passed which contains the Skolem substitutions to variables which are to apply. The key of an entry of this table is a Variable* and the value is a Term* which, internally is either a Constant* or a Function*. Whenever an existentially quantified formula or universally quantified formula gets the request to transform it into normal form, one of the following transformations are done:

- positive transformation:
  - **positive existential quantification:** $(\exists[X1,...,Xn] : F \equiv \exists[X1,...,Xn] : F)$
    create a new Skolem constant or function with the outer bound variables as arguments and add the substitution to the table.
  - **negative existential quantification:** $(\sim \exists[X1,...,Xn] : F \equiv \forall[X1,...,Xn] :\sim F)$
    Append the variables which are bound by this formula to the passed vector.
  - **positive universal quantification:** $(\forall[X1,...,Xn] : F)$
    Append the variables which are bound by this formula to the passed vector.
  - **negative universal quantification:** $(\sim \forall[X1,...,Xn] : F \equiv \exists[X1,...,Xn] :\sim F)$
    create a new Skolem constant or function with the outer bound variables as arguments and add the substitution to the table.
- negative transformation:
  - **positive existential quantification:** $(\sim \exists[X1,...,Xn] : F \equiv \forall[X1,...,Xn] :\sim F)$
    Append the variables which are bound by this formula to the passed vector.
  - **negative existential quantification:** $(\sim\sim \exists[X1,...,Xn] : F \equiv \exists[X1,...,Xn] : F)$
    create a new Skolem constant or function with the outer bound variables as arguments and add the substitution to the table.
  - **positive universal quantification:** $(\sim \forall[X1,...,Xn] : F \equiv \exists[X1,...,Xn] :\sim F)$
    create a new Skolem constant or function for all bound variables with the outer bound variables as arguments and add the substitution to the table.
  - **negative universal quantification:** $(\sim\sim \forall[X1,...,Xn] : F \equiv \forall[X1,...,Xn] : F)$
    Append the variables which are bound by this formula to the passed vector.

Actually, the behaviour of the skolemization can be inverted together with the polarities used for the normal form transformation by a command line parameter. Whenever a variable would be appended to the vector and whenever a skolemization would be done, the multiplicity (usage count) of the variable in the original formula structure is checked. If this is zero, the **variable is ignored** and thus is not used as argument for Skolem functions for example.

Note: Obviously, no prenex normal form is generated explicitly. But implicitly, it is. There are two concepts which assure that no variable scopes clash. If, for example an equivalence is transformed into a conjunction of disjunctions and the left or right side of the equivalence

contained a quantified formula, the quantified formula is transformed in both disjunctions. Since the bound variables vector and variable substitutions table are passed by value (they are copied), changes in the one branch of the transformation do not have effect to the other branch.[3] Furthermore, the parsing is adapted to construct formulae which are already in a form which is prenex-like. Whenever a quantified formula is visited while parsing, a new variable scope (table having the variable names as keys and the Variable objects as value) is created and appended to a vector and all variables which are bound by the quantified formula are inserted in this set. This insertion includes the creation of a new Variable object. Thus, if a variable is bound by multiple quantifications, they do have the same name but a different memory address. If a variable which is contained in a predicate or equality is parsed, a reverse iteration through the variable scope vector is done (from the end to the start of the vector). If the variable with the name is found in a variable scope, the Variable object is chosen. Thus, the Variable object of the most inner scope is used regardless whether a Variable object with the same object exists in an outer scope. Whenever a quantified formula was completely parsed, it's variable scope is removed from the variable scope vector which eliminates the possibility of using the Variable objects in a wrong scope.

If a variable name is not found in the vector, the variable is unbound and TEMPLAR can print a warning which can help the TPTP maintainers to identify ill-formed formulae.

In the following section, the unification in TEMPLAR will be described. The description includes the basis of the implementation and two variants which are implemented in TEMPLAR.

4.4 Unification

This section concerns the implementation of the unification as it is used by TEMPLAR. The basis is the concept of unification as it was described in the section 2.1 but the implementation is optimized for the way the formulae are represented in TEMPLAR. The unification algorithm which is used is based on the unification algorithm by Martelli and Montanari[MM82].

The basic algorithm which is described in the article by Martelli and Montanari can be described as follows.

Definition 10 (Basic Martelli-Montanari unification) Let $E = \{s_1 = t_1, ..., s_n = t_n\}$ with $i, n \in \mathbb{N}$ and $i \leq n$ be a set of equations where s_i is the top-level term of the one predicate and t_i the top-level term of the other predicate.

Apply one of the following rules until no rule can be applied any more.

1. $\neg var(s_i)$ and $var(t_i) \rightarrow E' = E \cup \{t_i = s_i\}$ (Swap)
2. $s_i == t_i \rightarrow$ remove $\{s_i = t_i\}$ (Identity elimination)
3. $s_i(x_1, x_2, ..., x_n)$ and $t_i(y_1, y_2, ..., y_n)$ and $s_i == t_i \rightarrow E' = E \cup \{x_j = y_j\}$ with $1 \leq j \leq n$ (Functional decomposition)

[3] This only holds for formulae which do not contain equivalences. Though the skolemization does work properly, the connection of one side of the transformed equivalence may lead to unwished restrictions for the other side. This is a conceptional bug which will be corrected in future development of TEMPLAR.

4. $var(s_i)$ and $\neg occurs(s_i, t_i) \rightarrow E' = E_{[s_i/t_i]}$ (Variable substitution)

When no rule can be applied any more, the unification is successful and the set E only contains equations whose left-hand sides are variables. If no rule can be applied but there are equations whose left-hand side is not a variable, the unification fails. The result of a successful unification is the most general unifier. □

This algorithm has, as stated in the article, a worst-case time complexity which is exponential. Though there is an algorithm with multi-equations and thus nearly linear complexity, this base algorithm is chosen since the use of pointers in TEMPLAR has a similar effect to multi-equations.

The version which is used is defined as follows.

Definition 11 (Adapted unification algorithm) Let $E = \{s_1 = t_1, ..., s_n = t_n\}$ with $i, n \in \mathbb{N}$ and $i \leq n$ be a set of equations where s_i is the top-level term of the one predicate and t_i the top-level term of the other predicate. Furthermore, let V be a variable substitution table with Variable pointers as key and Term pointers as value. This table was constructed in previous unifications.

First, apply all variable substitutions in V to the set E.

Let $type(Arg)$ be the type of a term which can be constant, variable or function and $==$ the equality of pointers or integers and $\neq$ the inequality. Also, let $\bowtie$ be the operator which applies the functional decomposition and $sub =< Variable*, Term* >$ be a temporary pair which saves a candidate substitution. Iterate over the set of Equations and apply the following rules in this order until no rule can be applied. If one sub-rule is applied, jump to the next equation. The result of every iteration is a set of equations I.

1. $s_i == t_i$, jump to next equation
2. $type(s_i) == type(t_i)$
   a) $type(s_i) == function : s_i.classifier == t_i.classifier \rightarrow I = I \cup (s_i \bowtie t_i)$, else fail
   b) $type(s_i) == variable$: if sub is not set, set it to $< s_i, t_i >$
3. $type(s_i) \neq type(t_i)$
   a) $type(t_i) == variable$:
   if $type(s_i) == function$ and t_i occurs in s_i fail if sub is not set, set it to $< t_i, s_i >$
   b) $type(s_i) == variable$:
   if $type(t_i) == function$ and s_i occurs in t_i fail if sub is not set, set it to $< s_i, t_i >$
   c) fail
4. if no functional decomposition was applied and sub is set: $I = I_{[sub.first/sub.second]}$
5. $E = I$

If this unification ends without failure, the set E contains the most general unifier. □

This unification has some positive aspects. The first rule instantly eliminates all constant and variable pairs which are equal as in the basic version of the unification. Extending the structure sharing to graph sharing (graphs of functions) would lead to the instant elimination of complete graphs.

The use of the candidate substitution is used in order to delay substitutions until no functional decomposition can be applied. This reduces the count of function copies which would be created otherwise. Also, the swap operation which is used in the basic algorithm is omitted here. The occurs-check has constant time complexity due to the fact that every function knows the variables which are contained. Also, pointer comparison has enormous advantages compared to string comparison. Since pointers are numbers, two pointers can be added binary (XOR) and the result has to be compared to zero. This are only two operations which can be done by the processor almost instantly.

Sometimes, no variables are contained (ground predicates) or only a syntactical comparison is needed. For this two cases, a special unification was implemented. It is similar to the normal unification but uses a different rule-set.

Definition 12 (Syntactical Equality) Let $E = \{s_1 = t_1, ..., s_n = t_n\}$ with $i, n \in \mathbb{N}$ and $i \leq n$ be a set of equations. Also, let $\bowtie$ be the operator which applies the functional decomposition. Iterate over the set of Equations and apply the following rules in this order until no rule can be applied. The result of every iteration is a set of equations I. Apply the following rules until E is empty.

1. $s_i == t_i$, jump to next equation
2. $type(s_i) == function$ and $type(t_i) == function : s_i.classifier == t_i.classifier \rightarrow I = I \cup (s_i \bowtie t_i)$, else fail

If the check terminates and E is empty, the two predicates are identical. □

The search engines are based on the implementation of the unification and the linguistic analysis. The following section introduces the implementation of the search engines and the way they are used.

4.5 Search Engines

The search engines are the most important parts of TEMPLAR but the implementation of them is almost equivalent to the descriptions in the chapter 3. Nevertheless, there are some details which are worth to be described.

First of all the search engines are implemented as modules which can be chosen by a command line parameter, as described in the workflow description. But to assure modularity and also make the implementation extendible, they need to have some common interfaces. They at least need to have the same functions for triggering the search iteration and for access to the currently selected formulae.

Thus, a class hierarchy was designed to ensure the common interfaces and also the easy extendibility of the set of search engines. This hierarchy is shown in figure 4.6.

The FormulaSelector class is the interface to the different selectors and provides the basic functionalities. All selectors need to implement this interface in order to be usable. While the ARDEFoFSelector (implementation of the basic selection concept) is a reimplementation of ARDE, the NonStructuralFoFSelector, the StructuralFoFSelector, the FrequencyBasedSelector and the DynamicSelector are new modules. The DynamicSelector is the default module which also has the highest performance. This is due to the fact that it inherits all functionalities from the StructuralFoFSelector and thus can access the strongest restrictions to the search graph. Furthermore, the

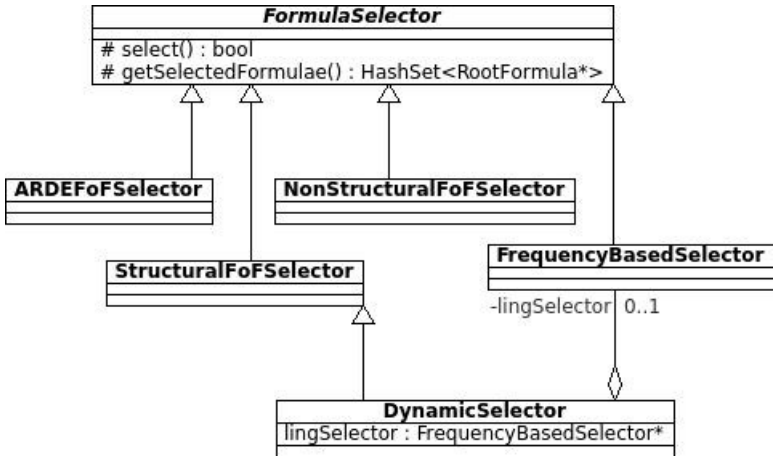

Figure 4.6: The class hierarchy of the search engines

DynamicSelector contains a FrequencyBasedSelector which uses the natural language processing approach. The DynamicSelector is more a scheduler which decides whether the frequency based selection should be used. Consider a big formula set of, let's say, 50.000 formulae. The probability is high that the theorem prover will not be able to solve the problem with the given formula set. Thus, the DynamicSelector triggers the frequency based selection. In the first *select* call, the DynamicSelector creates a beam vector by the frequency based selection. The beam vector contains vectors itself which contain the selected formula sets. Every entry of the beam vector except the last is created by triggering the frequency based selection for only one lexeme of the conjecture. If for example, the conjecture contains four lexemes whose tf-idf value is high enough to trigger the conjecture, five beams (vectors of formulae) are created. The last beam contains the merged formula sets of all other beams and thus represents a complete search for the conjecture. But since a beam needs to have an end, the DynamicSelector needs to decide, when it has to end the beam expansion which is just a sequence of *select* calls to the FrequencyBasedSelector. For this, a threshold is used which currently is 66 % of the formula set for smaller formula sets (up to 5000 formulae), 50 % for medium sized formula sets (between 5000 and 15000) and 10000 formulae for sets which contain more than 15000 formulae. If the beam reaches the threshold, no further selections for the current lexeme are done but the set which exceeded the threshold is taken as the end of the beam. After that, all sets in the beams are sorted by their size, duplicate sets are removed and the sets are enqueued. On every *getSelectedFormulae* call by the BatchProcessor or ProblemProcessor, the head of the queue is returned until the queue is empty. Before emptying the queue, the DynamicSelector saves the biggest formula set of it and triggers the structural selection with the set as new base set after emptying the queue.

In some cases, the frequency based selection is not able to create a beam which exceeds the beam threshold with the default threshold for the tf-idf threshold. In this case, the DynamicSelector modulates the threshold for the FrequencyBasedSelector by increasing it and trying to create a beam with the new threshold. But the threshold is limited to a maximum. If this maximum is reached, the DynamicSelector restarts with the complete formula set and uses the StructuralFoF-Selector for selection.

If the formula set has less then 2000 formulae, no frequency based selection is applied but the

structural graph search is used.

The structural selector currently uses most of the search graph restriction techniques. It uses the reconnection suppression, the ground connection target merging and the conjunction cut techniques including the conjunctive closure. Also, the structural selector has it's own graph representation for the search. The graph is constructed by a set of Connection objects which are chained and have the following information.

- The source Predicate object
- The source RootFormula object
- The target Predicate object
- The target RootFormula object
- The source Connection object(is undefined iff the connection is a root)
- A set of outgoing connections per contained predicate in the target RootFormula
- A set of visited Predicate objects
- A set of variable substitutions per RootFormula

With this information, all graph restrictions can be applied. If, for example, one connection becomes invalid since the target RootFormula cannot be refuted, all outgoing connections can be marked as invalid and the source RootFormulae can be checked for validity.

Finally, the possible output formats which are supported by TEMPLAR, are described. This describes both output usable for theorem proving and output usable for visualisation.

4.6 Output

The output is one of the most important parts of TEMPLAR since the selected formulae are needed to be given to a theorem prover in some way. Thus, all formulae can be written into output files. All output files which are meant for a theorem prover are written to a temporary folder. The output file name is the problem file name with the suffix *.depthY* where *Y* is the number of the iteration. The formulae are printed in TPTP syntax by default and can be printed in their original or the Skolem normal form.

But there is another way of printing the formulae which is the DOT[4] syntax. This is a language which represents definitions of graphs and can be transformed into images of many formats like Scalable Vector Graphic (SVG). Formerly, this output was meant for debugging purposes, for example for checking whether the structure sharing and the equality scope are properly created.

Actually, the RootFormula object can output itself in DOT syntax in two different forms; the original formula structure and the skolem normal form. To achieve this functionality, a *printDOT* functionality was implemented in the base classes of the formula hierarchy and specialised in all subclasses. But the DOT output is not limited to formulae. In fact, the connection graph of the

[4] http://www.graphviz.org/doc/info/lang.html

structural search can be printed in DOT, together with the formulae. This allows the visualisation of the complete graph search.

Apart from formula outputs, other informations can be printed, too. For example, TEMPLAR contains a logging implementation which is inspired by tools like syslog[5]. But those tools have much more functionalities than are needed (for example different output streams for different logging hierarchies). TEMPLAR uses the following log tags[6].

1. FATAL : Something really bad occurred which cannot be resolved, TEMPLAR needs to be stopped. E.g. Problem file does not exist
2. ERROR : Something bad occurred which cannot be resolved and can lead to undefined behaviour, TEMPLAR is stopped if the behaviour would be too destructive. E.g. non-existing axiom file.
3. WARNING : Something weird happened which breaks some standards. TEMPLAR can recover from the exception. E.g. a quantified variable is never used
4. INFO : General purpose informations like opening a file or importing 17 files, for example.
5. DEBUG : Information with more precision, importing axiom file x with y axioms
6. TimeInfo : Information about durations of specific processes like import or problem processing.

The level of informations to print can be set at compile time via the *TEMPLAR_LOG_LEVEL* variable:

- level 0 : DEBUG, INFO, WARNING, ERROR, FATAL
- level 1 : INFO, WARNING, ERROR, FATAL
- level 2 : WARNING, ERROR, FATAL
- level 3 : ERROR, FATAL

The time info is always logged and the default log level is 1. Also, the preamble for logs can be set via the variable *TEMPLAR_LOG_PROLOG* at compile time. The standard is the TPTP comment syntax ("% SZS") but changing the preamble to "//" makes sense when DOT output is needed since graphviz ignores standard C++ comments which are "//".

the following listing shows an excerpt of a sample output of TEMPLAR in combination with leanCoP.

```
1 % SZS [WARNING] /My/Path/HOM456+1.p:aTRIVu_ANDu_EXISTSu_THM :
    Ignoring unused variable X
2 % SZS [WARNING] /My/Path/HOM456+1.p:aTRIVu_FORALLu_ORu_THM :
    Ignoring unused variable A
3 ...
```

[5]Reference: man syslog , syslog is one of the standard logging mechanisms for linux based systems

[6]A tag is a marker with a special semantic

```
4  % SZS [WARNING] /My/Path/HOM456+1.p:aTRIVu_EXISTSu_IMPu_THM :
       Ignoring unused variable X for skolemization
5  % SZS [WARNING] /My/Path/HOM456+1.p:aTRIVu_EXISTSu_IMPu_THM :
       Ignoring unused variable X
6  % SZS [TimeInfo] Problem file Parsing duration: 0.47 sec (CPU-
       Time) , 0.478712 sec. RT
7  % SZS [TimeInfo] LOCAL Axiom file import duration: 0 sec (CPU-
       Time) , 5e-06 sec. RT
8  % SZS [INFO] Included 0 new LOCAL ( of 0 ) axiom files with 0
       formulae
9  % SZS [INFO] Using 1787 formulae for dynamic selection
10 % SZS [INFO] Trying to select for conjecture aEMPTYu_DELETE
11 % SZS [INFO] Starting Inference Engine on original formulae :
       1787
12 % SZS [INFO] Round  time      selected   of   percentage
13 % SZS [INFO] 0      0.011582    50          1787        2.79799
14 % SZS [INFO] Found proof!
15 % SZS [INFO] Found Proof in depth 0 with 50 formulae after
       1.08468 seconds
16 % SZS [INFO] Stopped selection after 1.5931 seconds
17 % SZS status Theorem for /My/Path/HOM456+1.p
18 % SZS [TimeInfo] Used time for /My/Path/HOM456+1.p 1.5 sec (CPU
       -Time) , 1.67798 sec. RT
19 % SZS status Ended for /My/Path/HOM456+1.p
20
21 % SZS [INFO] Clearing memory
22 % SZS [TimeInfo] Clearing memory 0.09 sec (CPU-Time) , 0.095781
        sec. RT
23 % SZS [INFO] Batch results:
24 % SZS [INFO] Count of proven/processed problems: 1
25 % SZS [INFO] Count of given up problems: 0
26 % SZS [INFO] Avg. proof/processing duration: 1.6783
27 % SZS [INFO] Batch file results:
28 % SZS [INFO] Count of proven/processed problems: 1
29 % SZS [INFO] Count of given up problems: 0
30 % SZS [INFO] Deleting file /My/Path/HOM456+1.p
31 % SZS [TimeInfo] Total Runtime: 1.64 sec (CPU-Time) , 1.82812
       sec. RT
```

Listing 4.5: Sample output of TEMPLAR showing the logging implementation

First, some warnings are shown which consider variables which are bound in a formula but not used. Also, it is shown whether this variable would be skolemized and in which formula of what file, the warning occurs. Some time information like problem file parsing and axiom file parsing durations and also problem processing durations are shown. The iterations are shown, too, which allows the visualisation of the restrictions when the output of open, closed and invalid paths is

done. Furthermore, after every batch and batch file, results about the count of proven problems, average proof times for the solved problems and the count of problems which could not be solved are printed.

Note: The logging information that the problem file was deleted does not mean the deletion of the file in the filesystem but the deletion of the information which were saved about the file in-memory. So all normal forms, original forms and analysis information are deleted after completing a batch. The status logs are tailored for the CADE ATP Systems Competition (CASC) and need to be present in order to make TEMPLAR automatically testable by a supervisor.

5. Related Work

Theorem provers do have problems with big formula sets and some complex formulae which is quite obvious since these two metrics can lead to complex and slow proof searches. Thus, it is just natural to apply relevance filtering techniques in order to reduce the formula sets or support the proof search with hints.

There are already many approaches to support theorem proving of which some are described here in order to give an insight to the state of the art. The techniques can be roughly partitioned into three classes which are frequency based relevance filters, the semantics based relevance filters and the unification based relevance filters. Different authors use different names for their techniques. Some call it "premise selection" (e.g. used for Naproche[CKKS10]), some call it "sine qua non" (essential condition, e.g. used for SinE[HV11]) and some call it relevance restriction (e.g. used for the approach described in [PY03]). In order to use a consistent description, this document uses the term "relevance filter" since all these systems use a relevance metric in order to filter(prune or reorder) the formula sets.

Frequency based relevance filters use the frequency of predicate and function (which includes constants) symbols in the formula set in order to either reduce the set by pruning it or reorder them by relevance. SinE for example, is a frequency based relevance filter which uses symbol frequencies to decide about the relevance of formulae. If a symbol is both contained in the problem and in an axiom (shared symbol), it is seen as relevant. This relevance is transitively extended by iterative deepening where for every last iteration, formulae are selected which share a symbol with an already selected formula. But SinE uses some sophisticated extensions like the triggers relation. This relation states that, if the frequency of a symbol is at most n times higher than the least common symbol in the formula set, it triggers the formula. This extension penalizes common symbols but does not ignore them completely and can lead to a fine-grained selection since different seed symbols can be chosen. TEMPLAR also uses frequencies, shared symbols and trigger-like relations in the FrequencyBasedSelector but does not use absolute frequencies like SinE does. The tf-idf weight is a relative weight which also takes the frequency of a symbol in one formula into account and uses normalizations in order to reduce aberrations in the weight due to extreme high frequencies in one formula.

Another frequency based relevance filter is described in [RPS09] and reorders axioms based on their relevance. The relevance is determined by checking, how many predicate or function symbols are shared by two axioms. After the formulae were reordered, Divvy selects a subset of a specific size, for example 50 % in the first iteration, 25 % in the second and 75 % in the third iteration. This selection has some common concepts with SinE but has the drawback that 25 % of, let's say 100.000 formulae still are 25.000 formulae. Thus, for really big formula sets, this approach has to be adapted. Though the variation of the percentage increases the probability of

selecting a formula set which is small enough to handle, the approach seems somehow weaker than the concept of SinE.

The Premise Selection Algorithm (PSA) which is used by the Naproche system [CKKS10] is a somehow hybrid approach which can use both frequency based relevance filtering and machine learning. With the information about a proof and specific proof steps for a conjecture returned by a theorem prover, the PSA aims to infer the probably needed formulae for later proofs. Internally, it creates a connection graph which represents the proof steps and contains the used formulae as nodes. The path between the conjecture and some arbitrary formula node is then used as distance and the average distance used for multiple proofs can be used to infer the probably needed distance for later proofs. The PSA can also use the frequency based relevance filtering technique which is also implemented in Divvy but the PSA mostly relies on having some successful proof attempts in order to use it's full power. TEMPLAR is potentially also able to calculate the average search depth for successful proofs and also use some comparable selection technique to the one of Naproche since the search in the StructuralFoFSelector is represented as graph.

The second relevance filtering approach is commonly called semantic selection and uses interpretations of formulae. Two implementations of this approach are described in [Pud07] and [SP07] (SRASS). Both approaches compute models for the given formulae and use the clause normal form but they differ in the workflow. While the concept of Petr Pudlák computes interpretations of the given formulae with a model finder, SRASS uses the concept of Divvy in order to reduce the formula set first.

The third relevance filtering approach is the unification based relevance filtering and was described in [PY03], for example. The basis for the relevance is the unifiability via resolving clauses. The concept searches for connections like TEMPLAR does but it strictly uses the conjunctive normal form and the workflow is quite resolution-like. Every connection is seen as a distance step and the transitive distance between the clauses contained in the conjecture and the ones of some arbitrary formula can be computed by this way. The concept has the drawback that creating the conjunctive normal form can lead to an explosion of the count of clauses and thus of the possible paths. Though TEMPLAR uses a similar approach in the ARDEFoFSelector, the NonStructuralFoFSelector and the StructuralFoFSelector, TEMPLAR does not need clause forms and thus does not need to cope with the formula growths. Also, TEMPLAR uses restriction approaches which are more tailored for the connection calculus (e.g. the conjunctive closure) but are not limited to the calculus.

6. Evaluation

This chapter concerns the evaluation of the concept and implementation of TEMPLAR. First, the evaluation setup is described and after that, benchmarks and the results of them are introduced.

6.1 Evaluation Setup

All tests which were done before or after the CASC-24[Sut13] were done on a compute node (jk-009) at the University of Potsdam.

The table 6.1 shows the system setup for the benchmarks before and after the competition in comparison to the specifications of the compute nodes of the CASC-J6[Sut12a] (2012) and the CASC-24 (2013)

Features	jk-009	CASC-J6	CASC-24
Cores	8 (2x4)	8 (2x4)	4 (1x4)
Core-speed	2,66 GHz	2,4 GHz	2,333 GHz
Hyper-Threading (HT) active	no	yes	no
RAM	16 GB	48 GB	12 GB
Kernel version	3.2 (Ubuntu 12.04)	3.2 (Debian 6)	2.6.29 (Fedora 11)
GCC version	4.6.3	4.4.5	4.4.1
SWI-Prolog version	5.10.04	unknown	unknown

Table 6.1: Comparison between the compute node specification of the last two CASCs and the one of the University of Potsdam

Both the memory and the number of cores were reduced significantly from CASC-J6 to CASC-24 and the operating system was a step backwards and the specifications of the nodes of the CASC-24 were much weaker than the specifications of jk-009. Thus, TEMPLAR had to be cross-compiled for Fedora 11 in order to even be able to run. And the lack of memory and cores had some other drawbacks. The low memory amount forced the deactivation of some heuristics like the conjunctive closure which was implemented but not tested. Also, when TEMPLAR starts six leanCoP instances, the probability is high that they slow each other down when not enough cores are existent. Thus, TEMPLAR was existent in two flavours, a CASC-variant and the original (daily) variant. They differ in the activated features and table 6.2 shows the most important differences.

The deactivation of the equality scope is not a real drawback since it was not even used. Since the nexus path union is not implemented, it was not active, obviously. But the deactivation of the

Features	TEMPLAR::leanCoP	TEMPLAR
Equality scope	no	yes
Conjunctive Closure	no	yes
Blocking Mutex	no(errors on competition systems)	yes

Table 6.2: Comparison between the two variants of TEMPLAR concerning the activated/implemented features

conjunctive closure is a drawback since it speeds-up the connection processing. The blocking mutexes are something which works out-of-the-box since mutexes are standard in the C++ standard library. But the blocking variant somehow lead to run-time errors on the competition systems and thus, the non-blocking variant of mutex locks was used. But this variant has the drawback that every part which wants to acquire a lock on the mutex, needs to do busy waiting until it gains the lock and thus uses up CPU time. This can slow down the overall system.

6.2 Results from CASC-24

TEMPLAR::leanCoP was delivered for the CASC in a specialized variant. In order to reduce memory consumption, the equality scope computation was deactivated since it was not used anyway. But also, the conjunctive closure computation was not yet tested in this state and thus was not active which has negative performance impact for the graph search. Also, the synchronisation of the mutexes had to be adapted in order to work properly on the quite ancient machines with operating systems from the year 2009 which do not support all C++ features well which lead to potentially lost time due to busy waiting(non-blocking mutexes). Furthermore, leanCoP itself was not changed since the CASC-J6.

Nevertheless, TEMPLAR::leanCoP did not get the last position in the LTB ranking (with 28 proofs and 27 of them with TEMPLAR-output formula sets) as in the CASC-J6 and also was able to beat iProver in the ISA category and E-KRHyper in all categories. But the system was also not able to gain a really a good position. Thus, the results of the CASC[1] were reviewed in order to find out whether the restrictions had worked at all. All problems which were proven by TEMPLAR::leanCoP were given to leanCoP in order to find out whether leanCoP would be able to solve them. First, the original strategy selection from the CASC was used which lead to the proof of three problems by leanCoP. But since the strategy selection could just have been inadequate, a complete benchmark with 19 strategies of leanCoP was done in order to get the potentially provable problems.

The used leanCoP strategies for the benchmark for the post-competition benchmark were

1. cut,comp(7)
2. conj,def,cut **and** noeq,conj,def,cut
3. conj,def **and** noeq,conj,def
4. nodef,scut,cut **and** noeq,nodef,scut,cut

[1] http://www.cs.miami.edu/~tptp/CASC/24/WWWFiles/Results.html

5. conj,nodef,cut **and** noeq,conj,nodef,cut
6. def,cut **and** noeq,def,cut
7. scut,cut **and** noeq,scut,cut
8. def,scut,cut **and** noeq,def,scut,cut
9. reo(12),def,scut,cut **and** reo(12),noeq,def,scut,cut
10. reo(40),conj,scut,cut **and** reo(40),noeq,conj,scut,cut

with a time limit of 15 seconds for each strategy. For convenience, the features of specific parts of the strategies are described but a complete description can be found in [Ott10]. While *cut* applies restricted backtracking in the proof search, *scut* applies this cut for the start clause. The option *conj* states, that clauses of the conjecture are taken as start clause. The option *noeq* removes equalities from the clauses and omits the insertion of the axioms of equality. While the option *nodef* forces a disjunctive normal form transformation for all formulae, the option *def* forces a definitional transformation. The option *reo(X)* forces a reordering of the clauses in the matrix X times and the option *comp(Y)* forces a complete proof search when the limit for the search depth (Y) is reached without finding a proof.

Running these strategies on the problems which were solved in the competition by TEMPLAR::leanCoP lead to the results in table 6.3 which is derived from the complete benchmark in table A.1 which can be found in the appendix. The table shows for each category of the LTB di-

Category	Proven	leanCoP@CASC-24	leanCoP-pot
HOL	12	2	6
ISA	12	0	6
MZR	4	1	2

Table 6.3: CASC-24: Comparison between the solved problems and the ratio of provable problems for leanCoP in CASC and potentially provable problems for leanCoP, excerpt of table A.1

vision the count of proven problems by TEMPLAR::leanCoP, the count of provable problems with the strategy chain for leanCoP at CASC and the count of potentially provable problems of leanCoP. As the table shows, leanCoP could potentially solve one half of the problems which were proven by TEMPLAR::leanCoP in CASC when the strategy chain would be chosen well. The table also shows that leanCoP would not be able to solve at least 14 of 28 problems. There must be a reason why so much problems become provable. Obviously, this must be due to the selected formulae. In order to show the strength of the restriction, the problems which were solved in CASC-24 were analysed with respect to the ratio of selected formulae. The table 6.4 shows a condensed variant of the tables A.2, A.3 and A.4 and in especial the counts of provable problems relative to the ratio of selected formulae (including the conjecture).

As the table shows, many (19/27) problems could be proven with less than one percent of the given formulae for the problem and all problems could be proven with less than 30 percent of the given formulae. The complete benchmarks in tables A.2, A.3 and A.4 also show that most of the problems could be solved with less than 100 formulae where at least 1000 and at most about 20000

Percentage	#proven
< 0.1	2
$0.1 - 0.5$	11
$0.51 - 1.0$	6
$1.1 - 10.0$	6
$20.1 - 29.9$	2

Table 6.4: Overview of selection ratios for the CASC-24 problems proven, extraction of tables A.2, A.3 and A.4

formulae were present. Also, the problem *ISD291* was provable by TEMPLAR::leanCoP with 24 of 19054 formulae after modulating the relevance threshold but the problem was not provable by leanCoP itself. Furthermore, no theorem prover except TEMPLAR::leanCoP was able to solve this problem in CASC.

The problem *MZS716* (a problem of the Mizar category) was solved with a set which was generated by the structural graph search on a set which was preselected by the linguistic search. Thus, there are cases where the coupling of graph search and linguistic selection works.

6.3 Results on problems of the CASC-J6

In order to analyse the performance of TEMPLAR compared to ARDE, all categories of the LTB division of the CASC-J6 were given to TEMPLAR. Precisely, TEMPLAR was tested on the Isabelle (ISA) category, the Mizar (MZR) category and the SUMO (SMO) category. Additionally, TEMPLAR was tested with the dynamic selection activated in combination with E. The used version of E was 1.6 which was the version that was used in the CASC-J6. All tests with E were done with the switch $--satauto$ which enables the integrated preprocessing of E but turns off the SinE selection.

> **Note:** The statistics module, which was implemented in TEMPLAR, generated the rows for all tables in this paragraph except for the table 6.8. Also, the full benchmark tables in the appendix were generated by TEMPLAR.

Results for the Isabelle category

First, the tests were done on the ISA category which contains 75 problems of which leanCoP-ARDE was able to solve 17 problems. Since the concept of ARDE was reimplemented in TEMPLAR with a quite fast unification algorithm and without use of Prolog bindings, the selection of the ARDE algorithm in TEMPLAR is generally faster than the former system. There was a time limit of 60 seconds per problem.

The table 6.5 shows the results for the benchmarks on the ISA category with the search engines which were implemented in TEMPLAR. The ARDE-selection, the structural and non-structural selection and the linguistic selection with the relevance thresholds 1.0 and 1.5 were tested. Also, the dynamic selection which starts a linguistic selection when more than 2000 (and more than 10) formulae are existent for a problem and modulates the relevance threshold was tested. The table shows per algorithm or algorithm variant the count of provable problems together with the

average proof duration including the preprocessing of TEMPLAR and the average ratio of selected formulae which was sufficient for a proof. The average ratio of selected formulae is calculated over the count of problems which were proven with TEMPLAR output files. The table is a condensed variant of the tables A.5, A.6, A.7, A.8, A.9, A.10, A.11, A.12 and A.13 which can be found in the appendix.

Algorithm	#proven	TEMPLAR	original	avg. selected	avg. duration
ARDE	14	11	3	47.184	8.1162
Non-Structural	13	8	5	31.4379	8.90493
Structural	15	10	5	40.6953	10.1525
Ling1.0	16	5	11	41.0158	13.5371
Ling1.5	19	14	5	31.7737	9.50811
Dynamic 2000	16	10	6	35.4682	12.7944
Dynamic 10	21	16	5	31.6782	18.1633
Dynamic 10-E	48	27	21	39.5659	14.1259

Table 6.5: Count of provable problems for the Isabelle category with the existent algorithms

As the table shows, the average ratio of selected formulae is smaller for the structural selection compared to the ARDE selection. Also, the average proof duration for the structural selection is only two seconds higher. Since the wall clock time is 60 seconds, a difference of two seconds is small. The count of provable formulae does not differ significantly for those two algorithms. The non-structural selection is weaker than the structural selection and the ARDE selection concerning the proofs found with TEMPLAR output. The linguistic selection is quite strong and leads to many proofs and even more proofs can be found compared to the structural selection. Obviously, the linguistic selection works for smaller formula sets, too. This is why the dynamic selection leads to less proofs compared to the linguistic selection variants. Only when the formula set is bigger than 2000, the embedded linguistic selection is activated. Thus, a dynamic selection which uses the linguistic selection when more than 10 formulae are present was tested, too. This selection leads to much more proofs compared to the structural search and linguistic selection but is a bit slower. Also, more than 66 % of the problems are proven with TEMPLAR output. The structural selection, on the other hand leads to the provability of problems which are not provable with the ARDE selection. For example, the problem ISA004 can be proven with the structural selection which chooses 497 of 533 given formulae. Using the dynamic selection with a formula limit of 10 together with the eprover (E) leads to 48 solvable problems of which more than 50 % are solved with TEMPLAR output.

Results for the SUMO category

The next benchmarks were done on the SUMO category which contains 20 problems of which leanCoP-ARDE was able to solve 2 problems. Every problem had a time limit of 60 seconds.

Again, the ARDE-selection, the non-structural and structural selection, the linguistic selection (in two variants), the dynamic selection (Since every problem had more than 2000 formulae, only the standard-variant was tested) and the dynamic selection coupled with E were tested and the results are shown in table 6.6. The table is a summarization of the tables A.14, A.15, A.16, A.17,

A.18, A.19 and A.20 which can be found in the appendix, too. Generally, neither leanCoP nor

Algorithm	#proven	TEMPLAR	original	avg. selected	avg. duration
ARDE	4	4	0	1.9601	7.92294
Non-Structural	4	4	0	1.9601	8.04163
Structural	4	4	0	1.9601	7.9368
Ling1.0	6	6	0	3.72015	11.9831
Ling1.5	6	6	0	4.13433	21.8285
Dynamic	7	7	0	2.86567	11.8458
Dynamic E	11	11	0	5.33094	11.1624

Table 6.6: Count of provable problems for the SUMO category with the existent algorithms

E are able to find a proof for the SUMO problems without preprocessing. Thus, both provers can benefit from the selection by TEMPLAR. All strategies lead to the provability of at least four problems. Even the reimplementation of ARDE leads to more proofs than the former ARDE implementation. The structural and non-structural selection lead to the same proofs and are comparable to ARDE concerning the average duration. The linguistic selection on the other hand is much stronger and both variants lead to six provable problems. Both variants lead to proofs which are not found with the other variant. The modulation of the relevance threshold which is implementation of the dynamic selection leads to the provability of seven problems for leanCoP and 11 problems for E.

Results for the Mizar category

The last benchmarks were done on the Mizar category which contains 80 problems in four batches with 20 problems each. Only the first batch the +1 problems were evaluated completely since the higher batches contain more problems and the memory consumption of TEMPLAR together with multiple instances of E or leanCoP was so high that the processes were stopped by the operating system. The first batch had a time limit of 30 seconds, the second 60 seconds, the third 90 seconds and the fourth 120 seconds per problem.

Since all problems of the first batch had less than 2000 formulae, the dynamic selection with a threshold of 10 formulae was used. The table 6.7 shows the results for this batch and is the condensation of the tables A.21, A.22, A.23, A.24, A.25, A.26 and A.27 which are located in the appendix.

leanCoP itself is potentially able to solve two of the problems with the original formula sets and the selection by TEMPLAR does not lead to additionally solvable problems. This does not hold for E. Almost 50 % of the problems were proven with the output of TEMPLAR.

Finally, the results in this section are summarised and compared to the results of the system ARDE. The results for the competition mode with ARDE 0.5.6 refer to the document [Fra12a] which is a description and evaluation of the concept of ARDE.

The table 6.8 summarises the tables 6.5, 6.6 and 6.7 together with informations about the categories (formula and problem count and time limits). The first block in the table contains the count of axioms (min., max.) per category, the time limit per problem in seconds and the count of problems. The second block shows the count of potentially provable problems per category for

Algorithm	#proven	TEMPLAR	original	avg. selected	avg. duration
ARDE	2	1	1	86.9565	1.67785
Non-Structural	2	0	2	x	3.44914
Structural	2	1	1	39.1304	3.28891
Ling1.0	1	1	0	43.6364	2.09269
Ling1.5	2	2	0	43.7418	10.8384
Dynamic 10	1	1	0	45.4545	6.59868
Dynamic 10 E	13	6	7	56.5428	4.19006

Table 6.7: Count of provable problems for the MZR (+1) category with the existent algorithms

	ISA	SMO	MZR1	MZR2	MZR3	MZR4
problem #	75	20	20	20	20	20
min. axiom #	518	55568	47	2951	10008	29085
max. axiom #	5249	55619	196	10434	19100	71136
Time (sec.)	60	60	30	60	90	120
Proofs on original files (potential)						
leanCoP	19	0	2	0	0	0
E	44	0	12	0	0	0
In competition (CASC-J6)						
leanCoP-ARDE	17	2	2	0	0	0
EP-LTB	47	15	11	2	6	6
In competition mode with ARDE 0.5.6						
leanCoP-ARDE	18(13)	4(4)	2(1)	0	0	1(1)
E-ARDE	40(15)	5(5)	11(6)	2(2)	2(2)	1(1)
In competition mode with TEMPLAR 0.8.5						
TEMPLAR::leanCoP	21(16)	7(7)	1(1)	0	0	OOM
TEMPLAR::E	48(27)	11(11)	13(6)	1(1),OOM	OOM	OOM

Table 6.8: Provable problems

leanCoP and E which is an excerpt from [Fra12a]. For leanCoP, a proof was attempted by using multiple strategies and giving all strategies a 30 seconds time limit (the same strategies as in the benchmarks of the section "Results from CASC-24"). For E, the time limit was the normal one. The third block shows the results from CASC-J6 for E and leanCoP-ARDE[2]. The fourth block shows the competition mode for leanCoP-ARDE and E-ARDE as it was evaluated in [Fra12a]. The numbers in the brackets denote the count of problems which were proven by ARDE output. The last block shows the summarised results of the current benchmarks with the current version of TEMPLAR where the numbers in brackets denote the count of proven problems by TEMPLAR output. OOM denotes that the system was stopped due to lack of memory.

Note: The results for EP-LTB are already very good since the SinE selection could be improved during more than two years and the thresholds could be optimised with exhaustive benchmarks. Beating a well-tested concept is obviously not really easy in a quite limited time.

[2] http://www.cs.miami.edu/~tptp/CASC/J6/WWWFiles/Results.html

Comparing the competition modes of TEMPLAR and ARDE, leanCoP can benefit both in the ISA category and the SMO category by using TEMPLAR. With TEMPLAR, more problems become provable due to the possibility of using hybrid selection algorithms. In both categories, additional problems become provable. E benefits from the new selection concepts, too. In the ISA category, eight more problems (even one more than EP-LTB could prove in CASC-J6) are proven and the count of problems which are proven on a restricted formula set is nearly doubled. In the SMO category, the count of provable problems was more than doubled by TEMPLAR in comparison with ARDE but the results are not yet as good as the results with SinE. In the Mizar category (+1 batch), two more proofs can be found for E.

7. Conclusion and Future Work

This section presents a small overview of the conception and implementation of TEMPLAR and also some ways to improve the system in the future. The effects of TEMPLAR are described and motives are described which lead to the way TEMPLAR was implemented.

7.1 Conclusion

Initially, the scope of TEMPLAR was to exclusively use a structural graph search for relevance pruning. During the implementation and parallel benchmarks, the concept proved to be working but not strong enough to use the pure graph search. Thus, many graph pruning concepts were implemented which improved the performance. But still, the graph search could not be improved to a level where it would still be a search and not a proof algorithm though there were still some concepts which could significantly improve the performance. Due to the limited time window of a thesis, the concepts were described but another approach was tried. Adapting the tf-idf based linguistic selection which is a well-evaluated concept, the performance of TEMPLAR and also the effects for the underlying theorem provers could be improved noticeably. The conception of the hybrid search strategy (dynamic selection) which chains the graph search with the linguistic selection lead to significant improvements.

Though no benchmarks could be done before CASC-24 and the conjunctive closure was not active, the CASC-results for TEMPLAR::leanCoP have shown that the described concept for pruning actually works and leads to improvements for theorem proving with large formula sets. Also it was shown that a decent frequency based selection with theoretical background can be applied and also may have strong effects to the selection. The evaluation shows that both the structural graph search and the linguistic selection and also the dynamic module can lead to a significant performance increase for the underlying theorem prover. Also, it was shown that the relevance filtering works for smaller formula sets, too and does not depend on the calculus which is used by the underlying theorem prover. Overall, the concept of TEMPLAR has shown to be much stronger than the concept of ARDE.

TEMPLAR can be used as a standalone system which processes some problems and just outputs filtered formula sets but it can also be used in combination with an arbitrary theorem prover. For this, an executor module has to be implemented which only includes three interface functions and a constructor which have to be implemented. Also, TEMPLAR can generate multiple visualisations like for the formula structure and the graph search. The grammar which is used for parsing input files (partially concurrently) has shown so far to be unambiguous and has the ability to generate formula graphs with restricted structure sharing. The graph search itself is based on strict Skolem

normal form and the normal form transformations generate usable metrics on the fly while some meta informations from clause forms are generated, too.

The structure of TEMPLAR is strictly modular which simplifies the extension of the system.

But there are still parts of TEMPLAR which can be improved and should be since it still is in beta stadium.

7.2 Future Work

This section gives some impressions about some concepts in TEMPLAR which can be improved.

Structure Sharing The structure sharing can be improved by extending the sharing to complete graphs. For example, the graph of a function may be shareable when a function occurs in different positions with syntactically identical graphs. Initial graph sharing can be applied during the normal form transformations and the sharing can be extended during unifications. Furthermore, the central ObjectVault is some kind of bottleneck. Every time a constant is parsed, the ObjectFactory queries the ObjectVault for this constant. This is okay when the constant is not yet existent. But after that, querying the Vault for the constant is a waste of time and may delay the initial query of other Factories for the constant or other constants. Thus, the ObjectFactory should cache all shared constructs locally after the first enquiry to the ObjectVault.

The structure sharing can furthermore be extended to support complete predicates when the predicates are ground (no variables are contained). This would lead to an unification for ground predicates which has a time complexity of O(1) and all predicates which are syntactically identical would be equivalent (with disregard of the polarity).

Unification via equalities The equality scope which was implemented can be used to use unification via equalities. For this, the equality closures could be used in order to check whether two functions (or constants) that are normally not unifiable would be with application of the equalities which enclose the two predicates which contain the functions.

Dynamic Selection/Structural Selection The structural selection currently only uses the formula structure to recognise invalid connections. Lifting the concept to a semi-proof search by applying constraints for the variable substitutions for outgoing connections could have enormous impact on the performance of TEMPLAR. This constraints would include that only connections which are consistent to other connections concerning the variable substitutions and together with the other connections show the unsatisfiability of the formula would be valid.

Currently, the dynamic module starts a prover thread for every generated formula set. But when the beam vectors are created (linguistic selection for each lexeme independently), it seems to make more sense to create a prover instance which processes one beam.

The breadth-first search has shown to be slow in some cases which is not a wished result. Thus, other search algorithms like cost-based searches should be evaluated. For this, heuristics are needed which give hints on the connections which should be expanded.

Memory Management The benchmarks with the Mizar batches have shown that the memory consumption in the graph search is significant. In order to reduce the memory consumption, all modules and associated structures (predicate vector for ARDE selection, ConnectionPath structure for non-structural search, for example) should be removed. Furthermore, the current memory management in not optimal since the problem file objects can only be disposed after a batch file was completed since race conditions can occur while deleting a function which should not be needed any more but is needed by an file which is about to be loaded. To solve this problem, a decent garbage collection mechanism should be implemented.

Search Graph Restrictions Currently, there are some search graph restrictions for the structural graph search like the conjunctive closures. But the nexus path union is not yet implemented. In order to allow the nexus path union, the connection graph representation needs to be improved in order to allow a Connection object to have multiple sources. This adoption also needs a rewrite of the recognition of invalid paths concerning the back-tracing to the first Connection which would still be valid

Learning From Proofs The implementation of TEMPLAR which contains the executors would theoretically allow the analysis of of successful proofs. By this, it could be attempted to recognise common inference steps (involved formulae) and optimal search depths for the relevance filtering after successful proofs which could be expanded to machine learning algorithms. For example, a successful proof, which contains all inference step could be tried to match to the search graph of TEMPLAR.

Improving the linguistic selection The linguistic selection was not tested exhaustively and the normalisation functions and thresholds are probably not optimal. Thus, TEMPLAR should be tested exhaustively on the complete classical first order logic problem set of the TPTP with different relevance thresholds and normalisations.

Enlarging the scope of TEMPLAR The initial scope of TEMPLAR was the reduction of big formula sets. But the implementation lead to a system which can be extended to a framework with an automatic mode. Implementing specialised modules which can be accessed by interactive theorem provers or automated theorem provers is possible with reasonable work. Extending TEMPLAR by arithmetic modules, for example could support the implementation of lean theorem provers which only contain the core of the calculus which is used and use specialised modules without having functionalities which are not even needed for the problem class which is considered. The source code of the provers could become smaller and more easily (formally) verifiable which is not easy for some theorem provers.

A. Benchmark Results

A.1 Post-CASC-24 Benchmarks

The table A.1 shows the problems of the CASC-24 which were proven by TEMPLAR::leanCoP and which are also provable by leanCoP on it's own. The second column shows for every problem,

Problem	Successful of 19	Solved by cut,comp(7)
HOM127+1	17	yes
HOM163+1	17	yes
MZS086+1	17	yes
ISC292+1	9	no
ISA182+1	7	no
ISA084+1	6	no
ISB753+1	6	no
ISB840+1	6	no
HON030+1	4	no
HOP175+1	4	no
ISB586+1	3	no
HOQ331+1	2	no
HON097+1	2	no
MZS278+1	1	no

Table A.1: Problems of CASC-24 which are provable by leanCoP with the count of successful strategies with a time limit of 15 s. per strategy

how many strategies of 19 were able to solve the problem in up to 15 seconds and the last column states whether the standard strategy is able to solve this problem in up to 15 seconds.

The table A.2 shows the problems of the HOL category which were proven by TEMPLAR::leanCoP in CAsC-24 in combination with the count of sufficient formula count for the proof and the formula count ratio.

Problem CASC/Real name	Sufficient	Original	Ratio (%)
HOL012/HOM163+1	90	1455	6,2
HOL042/HON030+1	14	2400	0,6
HOL045/HON097+1	8	2484	0,3
HOL105/HOO448+1	37	3900	0,95
HOL132/HOP175+1	193	4670	4,1
HOL139/HOP346+1	1275	4845	26,3
HOL154/HOP743+1	12	5253	0,2
HOL165/HOQ215+1	69	5735	1,2
HOL171/HOQ331+1	9	5859	0,2
HOL196/HOQ731+1	3	6318	0,1
HOL189/HOQ776+1	1266	6271	20,1

Table A.2: The ratio of sufficient selection of formulae for the CASC-24 HOL category

The table A.3 shows the problems of the Isabelle category which were proven by TEMPLAR::leanCoP in CAsC-24 in combination with the count of sufficient formula count for the proof and the formula count ratio.

Problem CASC/Real name	Sufficient	Original	Ratio (%)
ISA004/ISA084+1	56	7789	0,7
ISA005/ISA086+1	58	7800	0,7
ISA009/ISA182+1	63	7898	0,8
ISA030/ISA537+1	103	8796	1,2
ISA072/ISB586+1	42	10801	0,4
ISA078/ISB712+1	19	12514	0,2
ISA081/ISB753+1	48	12559	0,4
ISA087/ISB840+1	4	12651	0,03
ISA114/ISC292+1	13	13130	0,1
ISA148/ISC893+1	4	16907	0,02
ISA178/ISD284+1	28	19046	0,1
ISA179/ISD291+1	24	19054	0,1

Table A.3: The ratio of sufficient selection of formulae for the CASC-24 ISA category

The table A.4 shows the problems of the Mizar category which were proven by TEMPLAR::leanCoP in CASC-24 in combination with the count of sufficient formula count for the proof and the formula count ratio.

Problem CASC/Real name	Sufficient	Original	Ratio (%)
MZR019/MZS086+1	5	1713	0,3
MZR053/MZS230+1	19	2105	0,9
MZR066/MZS278+1	65	2187	2,97
MZR167/MZS716+1	122	3367	3,6

Table A.4: The ratio of sufficient selection of formulae for the CASC-24 MZR category

A.2 Full Benchmarks on CASC-J6 Problems

This section contains the full benchmarks on the CASC-J6 problems for the categories ISA, MZR and SMO.

> **Note:** All tables in this section were automatically generated by the statistics module of TEMPLAR. Only the caption and the label had to be adapted. Furthermore, the table of the provable problems for E with the dynamic selection had to be segmented.

A.2.1 ISA

The table A.5 shows the problems which are provable with applied ARDE selection. All problems which were proven with a depth of -1 were proven with the original problem file. All other problems were proven with a restricted formula set. The table shows the depth, the count of formulae needed for a successful proof, the percentage of needed formulae and the proof duration including all preprocessing by TEMPLAR.

Problem name	Proof depth	Formula count	of	Percentage	Proof duration
ISA001.p	-1	535	535	100	5.24331
ISA026.p	1	1040	1269	81.9543	6.57882
ISA027.p	1	1028	1199	85.7381	5.45143
ISA028.p	1	1034	1269	81.4815	4.46688
ISA029.p	-1	1288	1288	100	2.37488
ISA037.p	-1	1288	1288	100	4.50913
ISA052.p	1	603	5231	11.5274	3.23844
ISA053.p	1	602	5230	11.5105	3.17906
ISA054.p	0	15	5225	0.287081	3.26912
ISA056.p	0	348	5225	6.66029	4.30183
ISA057.p	1	556	5224	10.6432	3.36678
ISA058.p	1	2274	5209	43.6552	21.7347
ISA061.p	2	4836	5211	92.8037	22.9108
ISA062.p	2	4832	5209	92.7625	23.0016

Table A.5: Provable problems of the Isabelle category with application of the ARDE selection algorithm

The table A.6 shows the problems which are provable with applied non-structural selection. All problems which were proven with a depth of -1 were proven with the original problem file. All other problems were proven with a restricted formula set. The table shows the depth, the count of formulae needed for a successful proof, the percentage of needed formulae and the proof duration including all preprocessing by TEMPLAR.

Problem name	Proof depth	Formula count	of	Percentage	Proof duration
ISA001.p	-1	535	535	100	5.3714
ISA026.p	-1	1269	1269	100	6.86351
ISA027.p	1	1028	1199	85.7381	5.64025
ISA028.p	1	1034	1269	81.4815	4.7006
ISA029.p	-1	1288	1288	100	2.55604
ISA052.p	1	603	5231	11.5274	3.70807
ISA053.p	1	602	5230	11.5105	3.63379
ISA054.p	0	15	5225	0.287081	3.37094
ISA056.p	0	348	5225	6.66029	4.5828
ISA057.p	1	556	5224	10.6432	3.66434
ISA058.p	1	2274	5209	43.6552	22.1573
ISA061.p	-1	5211	5211	100	24.7959
ISA062.p	-1	5209	5209	100	24.7192

Table A.6: Provable problems of the Isabelle category with application of the non-structural graph search algorithm

The table A.7 shows the problems which are provable with applied structural selection. All problems which were proven with a depth of -1 were proven with the original problem file. All other problems were proven with a restricted formula set. The table shows the depth, the count of formulae needed for a successful proof, the percentage of needed formulae and the proof duration including all preprocessing by TEMPLAR.

Problem name	Proof depth	Formula count	of	Percentage	Proof duration
ISA001.p	-1	535	535	100	7.40128
ISA003.p	-1	533	533	100	11.5371
ISA004.p	1	497	533	93.2458	16.7922
ISA026.p	-1	1269	1269	100	7.62357
ISA027.p	1	1002	1199	83.5696	5.42609
ISA028.p	1	1018	1269	80.2206	4.4307
ISA029.p	-1	1288	1288	100	3.47324
ISA052.p	1	603	5231	11.5274	4.48867
ISA053.p	1	602	5230	11.5105	4.49673
ISA054.p	0	15	5225	0.287081	4.2486
ISA056.p	0	348	5225	6.66029	4.86342
ISA057.p	1	555	5224	10.624	4.51909
ISA058.p	-1	5209	5209	100	24.7756
ISA061.p	2	3056	5211	58.6452	24.1603
ISA062.p	2	2639	5209	50.6623	24.0514

Table A.7: Provable problems of the Isabelle category with application of the structural graph search algorithm

The table A.8 shows the problems which are provable with applied linguistic selection with a relevance threshold of 1.0. All problems which were proven with a depth of -1 were proven with the original problem file. All other problems were proven with a restricted formula set. The table shows the depth, the count of formulae needed for a successful proof, the percentage of needed formulae and the proof duration including all preprocessing by TEMPLAR.

Problem name	Proof depth	Formula count	of	Percentage	Proof duration
ISA001.p	-1	535	535	100	7.10718
ISA002.p	-1	533	533	100	13.3917
ISA004.p	3	370	533	69.4184	13.2984
ISA005.p	2	15	523	2.86807	2.22777
ISA026.p	-1	1269	1269	100	5.5175
ISA027.p	1	1029	1199	85.8215	3.56719
ISA028.p	-1	1269	1269	100	4.51504
ISA029.p	0	604	1288	46.8944	1.47902
ISA037.p	-1	1288	1288	100	40.3974
ISA052.p	1	4	5231	0.0764672	3.5283
ISA053.p	-1	5230	5230	100	6.64438
ISA054.p	-1	5225	5225	100	21.9984
ISA057.p	-1	5224	5224	100	23.0457
ISA058.p	-1	5209	5209	100	22.143
ISA061.p	-1	5211	5211	100	25.3163
ISA062.p	-1	5209	5209	100	22.4169

Table A.8: Provable problems of the Isabelle category, linguistic selection algorithm (relevance threshold:1.0)

The table A.9 shows the problems which are provable with applied linguistic selection with a relevance threshold of 1.5. All problems which were proven with a depth of -1 were proven with the original problem file. All other problems were proven with a restricted formula set. The table shows the depth, the count of formulae needed for a successful proof, the percentage of needed formulae and the proof duration including all preprocessing by TEMPLAR.

Problem name	Proof depth	Formula count	of	Percentage	Proof duration
ISA001.p	-1	535	535	100	6.465
ISA003.p	1	393	533	73.7336	10.8065
ISA005.p	0	15	523	2.86807	2.33939
ISA008.p	1	54	533	10.1313	5.99471
ISA009.p	2	16	519	3.08285	20.7621
ISA026.p	5	1165	1269	91.8046	5.61716
ISA027.p	2	1146	1199	95.5796	5.50898
ISA028.p	-1	1269	1269	100	4.4518
ISA029.p	-1	1288	1288	100	1.48661
ISA037.p	0	592	1288	45.9627	3.50603
ISA052.p	2	25	5231	0.47792	3.27052
ISA053.p	1	12	5230	0.229446	3.69588
ISA054.p	-1	5225	5225	100	22.0126
ISA057.p	3	3108	5224	59.4946	25.2081
ISA058.p	-1	5209	5209	100	22.1412
ISA059.p	1	21	5223	0.402068	3.87323
ISA060.p	0	26	5228	0.497322	3.97422
ISA061.p	1	7	5211	0.134331	4.08254
ISA062.p	1	3148	5209	60.4339	25.4575

Table A.9: Provable problems of the Isabelle category, linguistic selection algorithm (relevance threshold:1.5)

The table A.10 shows the problems which are provable with applied dynamic selection. All problems which were proven with a depth of -1 were proven with the original problem file. All other problems were proven with a restricted formula set. The table shows the depth, the count of formulae needed for a successful proof, the percentage of needed formulae and the proof duration including all preprocessing by TEMPLAR.

Problem name	Proof depth	Formula count	of	Percentage	Proof duration
ISA001.p	-1	535	535	100	7.44508
ISA003.p	1	497	533	93.2458	35.4
ISA004.p	1	497	533	93.2458	34.2271
ISA026.p	-1	1269	1269	100	9.31426
ISA027.p	1	1002	1199	83.5696	4.92568
ISA028.p	1	1018	1269	80.2206	3.91777
ISA029.p	-1	1288	1288	100	3.221
ISA037.p	-1	1288	1288	100	6.20423
ISA052.p	5	23	5231	0.439686	6.3582
ISA053.p	-1	5230	5230	100	9.24176
ISA054.p	1	17	5225	0.325359	4.73453
ISA057.p	-1	5224	5224	100	31.4708
ISA058.p	5	50	5209	0.959877	9.53446
ISA060.p	1	24	5228	0.459067	4.91965
ISA061.p	6	30	5211	0.575705	9.30201
ISA064.p	7	86	5241	1.64091	24.4936

Table A.10: Provable problems of the Isabelle category with application of the Dynamic selection (linguistic selection for more than 2000 formulae and modulation of relevance threshold)

The table A.11 shows the problems which are provable with applied dynamic selection. The linguistic selection was applied when more than 10 formulae were present. All problems which were proven with a depth of -1 were proven with the original problem file. All other problems were proven with a restricted formula set. The table shows the depth, the count of formulae needed for a successful proof, the percentage of needed formulae and the proof duration including all preprocessing by TEMPLAR.

Problem name	Proof depth	Formula count	of	Percentage	Proof duration
ISA001.p	-1	535	535	100	6.18278
ISA003.p	24	290	533	54.409	36.7015
ISA004.p	-1	533	533	100	52.8575
ISA005.p	0	14	523	2.67686	2.44861
ISA008.p	3	16	533	3.00188	19.6333
ISA009.p	4	17	519	3.27553	32.3676
ISA026.p	5	1155	1269	91.0165	10.695
ISA027.p	19	899	1199	74.9791	28.3805
ISA028.p	2	1018	1269	80.2206	5.06488
ISA029.p	-1	1288	1288	100	6.79771
ISA034.p	2	649	1264	51.3449	24.9838
ISA037.p	22	900	1288	69.8758	22.8243
ISA052.p	3	7	5231	0.133818	7.89816
ISA053.p	-1	5230	5230	100	10.5657
ISA054.p	1	17	5225	0.325359	5.47113
ISA057.p	5	3760	5224	71.9755	32.6253
ISA058.p	5	50	5209	0.959877	12.3525
ISA060.p	2	25	5228	0.478194	6.02319
ISA061.p	6	30	5211	0.575705	7.5634
ISA062.p	-1	5209	5209	100	25.5296
ISA064.p	6	84	5241	1.60275	24.4639

Table A.11: Provable problems of the Isabelle category with application of the Dynamic selection (linguistic selection for more than 10 formulae and modulation of relevance threshold)

The table A.12 shows the problems which are provable with applied dynamic selection for E. Only a part of the problems is shown in this table The linguistic selection was applied when more than 10 formulae were present. All problems which were proven with a depth of -1 were proven with the original problem file. All other problems were proven with a restricted formula set. The table shows the depth, the count of formulae needed for a successful proof, the percentage of needed formulae and the proof duration including all preprocessing by TEMPLAR.

Problem name	Proof depth	Formula count	of	Percentage	Proof duration
ISA001.p	-1	535	535	100	2.35377
ISA002.p	2	483	533	90.6191	2.23066
ISA003.p	23	282	533	52.9081	5.83819
ISA004.p	29	335	533	62.8518	11.2359
ISA005.p	-1	523	523	100	1.39779
ISA006.p	22	325	532	61.0902	10.2485
ISA008.p	3	16	533	3.00188	16.098
ISA009.p	4	17	519	3.27553	5.50066
ISA014.p	6	13	529	2.45747	6.39205
ISA015.p	5	17	529	3.21361	5.61416
ISA026.p	4	467	1269	36.8006	5.09835
ISA027.p	16	857	1199	71.4762	16.7338
ISA028.p	-1	1269	1269	100	5.51746
ISA029.p	13	274	1288	21.2733	14.7536
ISA030.p	-1	1200	1200	100	4.51866
ISA031.p	-1	1269	1269	100	6.5663
ISA032.p	-1	1214	1214	100	5.59729
ISA033.p	-1	1180	1180	100	2.43732
ISA034.p	2	649	1264	51.3449	28.4224
ISA035.p	-1	1264	1264	100	12.903
ISA038.p	4	288	1183	24.3449	9.85522
ISA040.p	-1	1200	1200	100	47.5938
ISA041.p	-1	1214	1214	100	25.1094
ISA051.p	-1	5248	5248	100	15.0639

Table A.12: Provable problems for TEMPLAR::E of the Isabelle category with application of the Dynamic selection (linguistic selection for more than 10 formulae and modulation of relevance threshold), Part 1

The table A.13 shows the problems which are provable with applied dynamic selection for E. Only a part of the problems is shown in this table The linguistic selection was applied when more than 10 formulae were present. All problems which were proven with a depth of -1 were proven with the original problem file. All other problems were proven with a restricted formula set. The table shows the depth, the count of formulae needed for a successful proof, the percentage of needed formulae and the proof duration including all preprocessing by TEMPLAR.

Problem name	Proof depth	Formula count	of	Percentage	Proof duration
ISA052.p	3	7	5231	0.133818	6.76896
ISA053.p	-1	5230	5230	100	15.46
ISA054.p	1	17	5225	0.325359	4.90162
ISA055.p	-1	5230	5230	100	16.9647
ISA056.p	-1	5225	5225	100	10.9298
ISA057.p	5	3760	5224	71.9755	8.72237
ISA058.p	5	50	5209	0.959877	10.7578
ISA059.p	-1	5223	5223	100	10.7068
ISA060.p	1	24	5228	0.459067	5.27422
ISA061.p	6	30	5211	0.575705	6.67208
ISA062.p	-1	5209	5209	100	16.8585
ISA063.p	3	1584	5228	30.2984	7.33661
ISA064.p	10	3139	5241	59.8932	20.8999
ISA065.p	7	3158	5228	60.4055	17.4888
ISA066.p	-1	5230	5230	100	46.2834
ISA067.p	-1	5222	5222	100	31.1865
ISA068.p	-1	5225	5225	100	30.4413
ISA069.p	-1	5250	5250	100	35.0878
ISA070.p	3	3069	5225	58.7368	10.5215
ISA071.p	7	3342	5230	63.9006	14.3308
ISA072.p	-1	5244	5244	100	31.5512
ISA073.p	10	3848	5244	73.3791	24.7246
ISA074.p	3	4305	5249	82.0156	9.06752
ISA075.p	6	4215	5232	80.5619	18.0278

Table A.13: Provable problems for TEMPLAR::E of the Isabelle category with application of the Dynamic selection (linguistic selection for more than 10 formulae and modulation of relevance threshold), Part 2

A.2.2 SMO

The table A.14 shows the problems which are provable with applied selection by the ARDE algorithm. All problems were proven with a restricted formula set. The table shows the depth, the count of formulae needed for a successful proof, the percentage of needed formulae and the proof duration including all preprocessing by TEMPLAR.

Problem name	Proof depth	Formula count	of	Percentage	Proof duration
SMO098+7.p	0	108	55590	0.19428	1.52318
SMO104+7.p	0	16	55592	0.0287811	1.55185
SMO108+7.p	0	4024	55611	7.23598	27.0507
SMO089+7.p	0	212	55592	0.38135	1.56603

Table A.14: Provable problems of the SUMO category with application of the ARDE selection algorithm

The table A.15 shows the problems which are provable with applied selection by the non-structural selection algorithm. All problems were proven with a restricted formula set. The table shows the depth, the count of formulae needed for a successful proof, the percentage of needed formulae and the proof duration including all preprocessing by TEMPLAR.

Problem name	Proof depth	Formula count	of	Percentage	Proof duration
SMO098+7.p	0	108	55590	0.19428	1.4795
SMO104+7.p	0	16	55592	0.0287811	1.57434
SMO108+7.p	0	4024	55611	7.23598	27.3072
SMO089+7.p	0	212	55592	0.38135	1.80551

Table A.15: Provable problems of the SUMO category with application of the non-structural selection algorithm

The table A.16 shows the problems which are provable with applied selection by the non-structural selection algorithm. All problems were proven with a restricted formula set. The table shows the depth, the count of formulae needed for a successful proof, the percentage of needed formulae and the proof duration including all preprocessing by TEMPLAR.

Problem name	Proof depth	Formula count	of	Percentage	Proof duration
SMO098+7.p	0	108	55590	0.19428	1.52061
SMO104+7.p	0	16	55592	0.0287811	1.58818
SMO108+7.p	0	4024	55611	7.23598	27.0535
SMO089+7.p	0	212	55592	0.38135	1.58489

Table A.16: Provable problems of the SUMO category with application of the structural selection algorithm

The table A.17 shows the problems which are provable with applied selection by the linguistic selection algorithm with a relevance threshold of 1.0. All problems were proven with a restricted formula set. The table shows the depth, the count of formulae needed for a successful proof, the

percentage of needed formulae and the proof duration including all preprocessing by TEMPLAR.

Problem name	Proof depth	Formula count	of	Percentage	Proof duration
SMO098+7.p	0	1272	55590	2.28818	5.37817
SMO104+7.p	0	5	55592	0.0089941	4.62122
SMO089+7.p	0	36	55592	0.0647575	4.13798
SMO105+7.p	1	1716	55592	3.08678	23.1374
SMO106+7.p	1	8096	55594	14.5627	25.7506
SMO107+7.p	2	1284	55597	2.30948	8.87298

Table A.17: Provable problems of the SUMO category, linguistic selection algorithm (relevance threshold:1.0)

The table A.18 shows the problems which are provable with applied selection by the linguistic selection algorithm with a relevance threshold of 1.5. All problems were proven with a restricted formula set. The table shows the depth, the count of formulae needed for a successful proof, the percentage of needed formulae and the proof duration including all preprocessing by TEMPLAR.

Problem name	Proof depth	Formula count	of	Percentage	Proof duration
SMO098+7.p	0	4109	55590	7.39162	27.1641
SMO104+7.p	0	9	55592	0.0161894	23.0257
SMO108+7.p	1	13	55611	0.0233767	15.6477
SMO089+7.p	0	83	55592	0.149302	20.7206
SMO105+7.p	1	9565	55592	17.2057	28.439
SMO107+7.p	0	11	55597	0.0197852	15.9741

Table A.18: Provable problems of the SUMO category, linguistic selection algorithm (relevance threshold:1.5)

The table A.19 shows the problems which are provable with applied dynamic selection. All problems which were proven with a depth of -1 were proven with the original problem file. All other problems were proven with a restricted formula set. The table shows the depth, the count of formulae needed for a successful proof, the percentage of needed formulae and the proof duration including all preprocessing by TEMPLAR.

Problem name	Proof depth	Formula count	of	Percentage	Proof duration
SMO098+7.p	0	4	55590	0.00719554	6.22771
SMO104+7.p	0	3	55592	0.00539646	7.78835
SMO108+7.p	3	13	55611	0.0233767	5.95732
SMO089+7.p	0	36	55592	0.0647575	6.03142
SMO105+7.p	1	1716	55592	3.08678	23.9873
SMO106+7.p	1	8096	55594	14.5627	26.5442
SMO107+7.p	2	1284	55597	2.30948	6.38424

Table A.19: Provable problems of the SUMO category with application of the Dynamic selection (linguistic selection for more than 2000 formulae and modulation of relevance threshold)

The table A.20 shows the problems which are provable with applied dynamic selection for E. All problems which were proven with a depth of -1 were proven with the original problem file. All other problems were proven with a restricted formula set. The table shows the depth, the count of formulae needed for a successful proof, the percentage of needed formulae and the proof duration including all preprocessing by TEMPLAR.

Problem name	Proof depth	Formula count	of	Percentage	Proof duration
SMO098+7.p	0	4	55590	0.00719554	5.22823
SMO104+7.p	0	3	55592	0.00539646	7.75883
SMO108+7.p	3	13	55611	0.0233767	5.36057
SMO089+7.p	0	36	55592	0.0647575	6.07582
SMO105+7.p	1	1716	55592	3.08678	4.88345
SMO106+7.p	1	8096	55594	14.5627	15.9235
SMO086+6.p	6	5640	55588	10.1461	24.5339
SMO109+7.p	4	1582	55589	2.84589	6.53374
SMO107+7.p	2	1284	55597	2.30948	5.30074
SMO075+6.p	4	9077	55588	16.3291	33.3223
SMO093+7.p	5	5150	55618	9.25959	7.86525

Table A.20: Provable problems for TEMPLAR::E of the SUMO category with application of the Dynamic selection (linguistic selection for more than 10 formulae and modulation of relevance threshold)

A.2.3 MZR

The table A.21 shows the problems which are provable with applied ARDE selection. All problems which were proven with a depth of -1 were proven with the original problem file. All other problems were proven with a restricted formula set. The table shows the depth, the count of formulae needed for a successful proof, the percentage of needed formulae and the proof duration including all preprocessing by TEMPLAR.

Problem name	Proof depth	Formula count	of	Percentage	Proof duration
LAT288+1.p	1	60	69	86.9565	1.16509
SEU449+1.p	-1	55	55	100	2.19062

Table A.21: Provable problems of the Mizar (+1) category with application of the ARDE selection algorithm

The table A.22 shows the problems which are provable with applied non-structural selection. All problems which were proven with a depth of -1 were proven with the original problem file. All other problems were proven with a restricted formula set. The table shows the depth, the count of formulae needed for a successful proof, the percentage of needed formulae and the proof duration including all preprocessing by TEMPLAR.

Problem name	Proof depth	Formula count	of	Percentage	Proof duration
LAT288+1.p	-1	69	69	100	2.41537
SEU449+1.p	-1	55	55	100	4.48292

Table A.22: Provable problems of the Mizar (+1) category with application of the non-structural graph search algorithm

The table A.23 shows the problems which are provable with applied structural selection. All problems which were proven with a depth of -1 were proven with the original problem file. All other problems were proven with a restricted formula set. The table shows the depth, the count of formulae needed for a successful proof, the percentage of needed formulae and the proof duration including all preprocessing by TEMPLAR.

Problem name	Proof depth	Formula count	of	Percentage	Proof duration
LAT288+1.p	0	27	69	39.1304	2.66848
SEU449+1.p	-1	55	55	100	3.90934

Table A.23: Provable problems of the Mizar (+1) category with application of the structural graph search algorithm

The table A.24 shows the problems which are provable with applied selection by the linguistic selection algorithm with a relevance threshold of 1.0. All problems were proven with a restricted formula set. The table shows the depth, the count of formulae needed for a successful proof, the percentage of needed formulae and the proof duration including all preprocessing by TEMPLAR.

Problem name	Proof depth	Formula count	of	Percentage	Proof duration
SEU449+1.p	0	24	55	43.6364	2.09269

Table A.24: Provable problems of the Mizar (+1) category, linguistic selection algorithm (relevance threshold:1.0)

The table A.25 shows the problems which are provable with applied selection by the linguistic selection algorithm with a relevance threshold of 1.5. All problems were proven with a restricted formula set. The table shows the depth, the count of formulae needed for a successful proof, the percentage of needed formulae and the proof duration including all preprocessing by TEMPLAR.

Problem name	Proof depth	Formula count	of	Percentage	Proof duration
LAT288+1.p	0	29	69	42.029	19.6021
SEU449+1.p	0	25	55	45.4545	2.07462

Table A.25: Provable problems of the Mizar (+1) category, linguistic selection algorithm (relevance threshold:1.5)

The table A.26 shows the problems which are provable with applied dynamic selection. All problems which were proven with a depth of -1 were proven with the original problem file. All other problems were proven with a restricted formula set. The table shows the depth, the count of formulae needed for a successful proof, the percentage of needed formulae and the proof duration including all preprocessing by TEMPLAR.

Problem name	Proof depth	Formula count	of	Percentage	Proof duration
SEU449+1.p	6	25	55	45.4545	6.59868

Table A.26: Provable problems for TEMPLAR::E of the Mizar +1 category with application of the Dynamic selection (linguistic selection for more than 2000 formulae and modulation of relevance threshold)

The table A.27 shows the problems which are provable with applied dynamic selection for E. All problems which were proven with a depth of -1 were proven with the original problem file. All other problems were proven with a restricted formula set. The table shows the depth, the count of formulae needed for a successful proof, the percentage of needed formulae and the proof duration including all preprocessing by TEMPLAR.

Problem name	Proof depth	Formula count	of	Percentage	Proof duration
ALG220+1.p	0	9	69	13.0435	1.5936
GRP637+1.p	-1	55	55	100	2.22519
GRP639+1.p	-1	54	54	100	2.01231
LAT288+1.p	-1	69	69	100	2.01223
SEU451+1.p	-1	48	48	100	13.4015
SEU449+1.p	-1	55	55	100	2.81355
LAT302+1.p	-1	92	92	100	2.1552
LAT311+1.p	11	52	101	51.4851	2.97511
LAT343+1.p	-1	97	97	100	2.36519
SEU447+1.p	10	31	60	51.6667	1.95793
SEU441+1.p	14	46	61	75.4098	10.7585
TOP042+1.p	3	104	122	85.2459	2.70389
TOP032+1.p	6	83	133	62.406	7.49658

Table A.27: Provable problems for TEMPLAR::E of the Mizar +1 category with application of the Dynamic selection (linguistic selection for more than 10 formulae and modulation of relevance threshold)

B. Files of the thesis

The compact disc which is bundled with this thesis contains the following folder structure:

- **/src/** : the source code folder of TEMPLAR
- **/binary/** : pre-built binaries folder
- **/leanCoP/** : the version 2.2 of leanCoP
- **/tests/** : folder containing some test files for TEMPLAR
- **/thesis/** : folder containing TeX files, images, listings and benchmark results of this thesis
- **/README** : a readme file which describes the usage of TEMPLAR
- **TEMPLAR.pdf** : this document in digital format

List of Figures

List of Tables

Listings

C. Abbreviations

CASC CADE ATP Systems Competition

CNF Conjunctive Normal Form

EBNF Extended Backus-Naur Form

ESC Electronic Stability Control

FIFO First In First Out

fof first order formula

fol first order logic

HT Hyper-Threading

iff if and only if

LTB Large Theorem Batch

mutex mutual exclusion

mgu most general unifier

NLP Natural Language Processing

PSA Premise Selection Algorithm

UML Unified Modelling Language

SVG Scalable Vector Graphic

TPTP Thousands of Problems for Theorem Provers

TEMPLAR TEMpestuous Pruner based on Logical Axiom Relevance

Bibliography

[BBJ07] G.S. Boolos, J.P. Burgess, and R.C. Jeffrey. *Computability and Logic*. Cambridge University Press, 2007. Available from: http://books.google.de/books?id=8ABhQgAACAAJ.

[BEL01] Matthias Baaz, Uwe Egly, and Alexander Leitsch. Normal form transformations. In John Alan Robinson and Andrei Voronkov, editors, *Handbook of Automated Reasoning*, pages 273–333. Elsevier and MIT Press, 2001.

[Bib92] Wolfgang Bibel. *Deduktion: Automatisierung der Logik*. Oldenbourg, 1992.

[Bir10] Richard Bird. *Pearls of Functional Algorithm Design*. Cambridge University Press, 2010. Available from: http://www.cambridge.org/gb/knowledge/isbn/item5600469.

[CKKS10] Marcos Cramer, Peter Koepke, Daniel Kühlwein, and Bernhard Schröder. Premise selection in the naproche system. In *Proceedings of the 5th international conference on Automated Reasoning*, IJCAR'10, pages 434–440, Berlin, Heidelberg, 2010. Springer-Verlag. Available from: http://dx.doi.org/10.1007/978-3-642-14203-1_37.

[dlT92] Thierry Boy de la Tour. An optimality result for clause form translation. *J. Symb. Comput.*, 14(4):283–302, 1992. Available from: http://dblp.uni-trier.de/db/journals/jsc/jsc14.html#Tour92.

[Ede92] Elmar Eder. *Relative complexities of first order calculi*. Verlag Vieweg, Wiesbaden, Germany, Germany, 1992.

[Fra12a] Mario Frank. Axiom Relevance Decision Engine. Technical report, 2012. Available from: http://nbn-resolving.de/urn:nbn:de:kobv:517-opus-72128.

[Fra12b] Mario Frank. Relevanzbasiertes Preprocessing für automatische Theorembeweiser. In Johannes Schmidt, Thomas Riechert, and Sören Auer, editors, *SKIL 2012 – Dritte Studentenkonferenz Informatik Leipzig 2012*, volume XXXIV of *Leipziger Beiträge zur Informatik*, pages 87–98. Leipziger Informatik-Verbund (LIV), 2012. Klaus-Peter Fähnrich (Series Editor).

[Ger96] Michael Gervers. Malcolm barber, the new knighthood: A history of the order of the temple. cambridge, eng.: Cambridge university press, 1994. pp. xxi, 441; 17 black-and-white plates and 14 figures. $69.95. *Speculum*, 71:679–681, 6 1996. Available from: http://journals.cambridge.org/article_S003871340013413X.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns – Elements of Reusable Object-Oriented Software*. Addison-Wesley Longman, Amsterdam, 1 edition, 1995. 37. Reprint (2009).

[HKV11] Krystof Hoder, Laura Kovács, and Andrei Voronkov. Invariant generation in vampire. In Parosh Aziz Abdulla and K. Rustan M. Leino, editors, *TACAS*, volume 6605 of *Lecture Notes in Computer Science*, pages 60–64. Springer, 2011.

[HV11] Kryštof Hoder and Andrei Voronkov. Sine qua non for large theory reasoning. In *Proceedings of the 23rd international conference on Automated deduction*, CADE'11, pages 299–314, Berlin, Heidelberg, 2011. Springer-Verlag. Available from: http://dl.acm.org/citation.cfm?id=2032266.2032289.

[Jon72] Karen Spärck Jones. A statistical interpretation of term specificity and its application in retrieval. *Journal of Documentation*, 28:11–21, 1972.

[Kor08] K. Korovin. iProver – an instantiation-based theorem prover for first-order logic (system description). In A. Armando, P. Baumgartner, and G. Dowek, editors, *Proceedings of the 4th International Joint Conference on Automated Reasoning, (IJCAR 2008)*, volume 5195 of *Lecture Notes in Computer Science*, pages 292–298. Springer, 2008.

[MB72] J. Strother Moore and Robert S. Boyer. The sharing of structure in theorem-proving programs. volume 7, pages 101–116. Edinburgh University Press, 1972.

[MM82] Alberto Martelli and Ugo Montanari. An efficient unification algorithm. *TRANSACTIONS ON PROGRAMMING LANGUAGES AND SYSTEMS (TOPLAS)*, 4(2):258–282, 1982.

[OB03] Jens Otten and Wolfgang Bibel. leancop: lean connection-based theorem proving. *Journal of Symbolic Computation*, 36(1-2):139–161, 2003.

[Ott08] Jens Otten. leancop 2.0 and ileancop 1.2: High performance lean theorem proving in classical and intuitionistic logic (system descriptions). In *Proceedings of the 4th international joint conference on Automated Reasoning*, IJCAR '08, pages 283–291, Berlin, Heidelberg, 2008. Springer-Verlag. Available from: http://dx.doi.org/10.1007/978-3-540-71070-7_23.

[Ott10] Jens Otten. Restricting backtracking in connection calculi. *AI Commun.*, 23(2-3):159–182, April 2010. Available from: http://dl.acm.org/citation.cfm?id=1735921.1735931.

[Pud07] Petr Pudlak. Semantic selection of premisses for automated theorem proving. In Geoff Sutcliffe, Josef Urban, and Stephan Schulz, editors, *ESARLT*, volume 257 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2007.

[PW07] Björn Pelzer and Christoph Wernhard. System description: E-krhyper. In Frank Pfenning, editor, *CADE*, volume 4603 of *Lecture Notes in Computer Science*, pages 508–513. Springer, 2007. Available from: http://dblp.uni-trier.de/db/conf/cade/cade2007.html#PelzerW07.

[PY03] David A. Plaisted and Adnan H. Yahya. A relevance restriction strategy for automated deduction. *Artif. Intell.*, 144(1-2):59–93, 2003. Available from: http://dblp.uni-trier.de/db/journals/ai/ai144.html#PlaistedY03.

[RPS09] Alex Roederer, Yury Puzis, and Geoff Sutcliffe. Divvy: An atp meta-system based on axiom relevance ordering. In Renate A. Schmidt, editor, *CADE*, volume 5663 of *Lecture Notes in Computer Science*, pages 157–162. Springer, 2009.

[Rus08] Bertrand Russell. Mathematical logic as based on the theory of types. *American Journal of Mathematics*, 30(3):222–262, 1908. Available from: http://links.jstor.org/sici?sici=0002-9327%28190807%2930%3A3%3C222%3AMLABOT%3E2.0.CO%3B2-G,http://www.cfh.ufsc.br/~dkrause/pg/cursos/selecaoartigos/Russell(1905).pdf.

[Sch04] S. Schulz. System Description: E 0.81. In D. Basin and M. Rusinowitch, editors, *Proc. of the 2nd IJCAR, Cork, Ireland*, volume 3097 of *LNAI*, pages 223–228. Springer, 2004.

[Sip96] Michael Sipser. *Introduction to the Theory of Computation*. International Thomson Publishing, 1st edition, 1996.

[Smu95] R.M. Smullyan. *First-order logic*. Dover Publications, 1995.

[SP07] Geoff Sutcliffe and Yury Puzis. Srass - a semantic relevance axiom selection system. In Frank Pfenning, editor, *CADE*, volume 4603 of *Lecture Notes in Computer Science*, pages 295–310. Springer, 2007.

[Sut09] G. Sutcliffe. The TPTP Problem Library and Associated Infrastructure: The FOF and CNF Parts, v3.5.0. *Journal of Automated Reasoning*, 43(4):337–362, 2009.

[Sut12a] G. Sutcliffe. The CADE-23 Automated Theorem Proving System Competition - CASC-23. *AI Communications*, 25(1):49–63, 2012.

[Sut12b] Geoff Sutcliffe, editor. *CASC-J6 Proceedings*, volume 11 of *EPiC Series*. EasyChair, 2012.

[Sut13] G. Sutcliffe. Proceedings of the 24th CADE ATP System Competition. Lake Placid, USA, 2013.

[Tan07] Andrew S. Tanenbaum. *Modern Operating Systems*. Prentice Hall Press, Upper Saddle River, NJ, USA, 3rd edition, 2007.

[USPV08] Josef Urban, Geoff Sutcliffe, Petr Pudlák, and Jirí Vyskocil. Malarea sg1- machine learner for automated reasoning with semantic guidance. In Alessandro Armando, Peter Baumgartner, and Gilles Dowek, editors, *IJCAR*, volume 5195 of *Lecture Notes in Computer Science*, pages 441–456. Springer, 2008.